A SOFTWARE DEFINED NETWORKING APPROACH FOR LINK FAILURE RECOVERY IN CRITICAL NETWORK INFRASTRUCTURES

by

Shaikhum Monira
Exam Roll: Curzon Hall-254
Registration No: 2012-31-2009
Session: 2012-2013

Minhaz Raufoon
Exam Roll: Curzon Hall-224
Registration No: 2012-01-2020
Session: 2012-2013

4th year (B.Sc. Honors)

A project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science (B.Sc.) in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
UNIVERSITY OF DHAKA

February 2017

Declaration

We do hereby declare that the works presented in this project titled "A Software Defined Networking Approach for Fast Failover in Critical Network Infrastructures" is an original work solely performed by ourselves under the supervision of Dr. Syed Faisal Hasan, Associate Professor, Department of Computer Science and Engineering, University of Dhaka. We also declare that no part of this project is plagiarized or has been submitted or published anywhere for the award of any degree or diploma.

Countersigned:

(Dr. Syed Faisal Hasan)
Associate Professor
Department of Computer Science and Engineering
University of Dhaka
Project Supervisor

Signatures:

(Shaikhum Monira)
Candidate

(Minhaz Raufoon)
Candidate

Acknowledgments

First of all, we express our gratitude to the Almighty who offered us His divine blessings, strength and patience to complete the works of this project.

It is our pleasure to acknowledge the extraordinary supervision, invaluable suggestions and proper guidance of our respectable teacher and thesis supervisor Dr. Syed Faisal Hasan, Associate Professor, Department of Computer Science and Engineering, University of Dhaka. Without his kind cooperation and help, the overall project would never be possible. We consider ourselves very fortunate for being able to work with a considerate and encouraging mentor like him. We express our humble gratitude to him for his affectionate guidance.

We are indebted to all of our teachers at the Department of Computer Science and Engineering, University of Dhaka for their valuable suggestions and supports. A special thanks goes to our teacher Muhammed Tawfiqul Islam for his intellectual support, useful materials and important suggestions.

Lastly, we are thankful to our parents, friends and well-wishers for their endless supports and encouragements which have always kept us inspired and motivated.

Abstract

Often, service disruption due to communication network's link failure is a reality due to unavoidable circumstances. When such failures occur in critical networks like the smart power grid, transportation, it is essential to recover from these link failures in order to avoid major catastrophes. The existing link failure mechanisms of traditional networks are not efficient enough to ensure quick restoration of the path of the network packets to keep critical networks uninterrupted. In order to make the critical networks functional without noticeable delay and performance degradation, in this project we evaluated existing solutions and proposed a better approach. Our proposed solution leveraged the fast failover mechanism of Software Defined Networking (SDN) which is a new paradigm in networking. We also compared the performance of our proposed failover recovery mechanism with the existing link failure handling mechanisms to show its efficiency over these algorithms.

Contents

List of Figures

Chapter 1

Introduction

The use of computer networks in electricity generation, water supply, public health, financial services, security services etc. has made these infrastructures more manageable for distribution, remote controlling and monitoring. These infrastructures are termed as critical infrastructures where the communication delay of even a few milliseconds can degrade these services to a great extent. An example of critical infrastructure is the smart grid technology, where electrical grids are facilitated with computer network based remote control and monitoring services. It is crucial to maintain the performance and quality of service (QoS) of critical networks which includes availability, security, throughput and minimum response time. However, some obstacles in the performance of computer networks like link failure, link congestion are unavoidable.

Link failure is a common phenomenon in computer networks which may occur anytime due to network configuration change, network device error, power outage or many other reasons and results in the disruption of service distribution. This disruption results in unimaginable losses in critical networks. Thus, efficient mechanisms should be present to handle link failures to keep the flow of those networks uninterrupted. For handling link failure, network systems keep redundant links for each path so that in case of link failure the network systems may calculate a new backup path and keep the system uninterrupted. Keeping redundant links may create loops in computer networks. Traditional networks use Spanning Tree Protocol (STP) and Rapid Spanning Tree Protocol (RSTP) to maintain redundant links for handling link failure without creating loops in the networks where STP and RSTP take 30-50 seconds and approximately 10 seconds respectively to restore the failed path. Neither STP nor RSTP provides any efficient solution for highly time-sensitive critical network infrastructures which must respond immediately after getting a service request.

To overcome this limitation of traditional networks in handling link failure of critical networks, one possible solution can be Software Defined Network (SDN) which is a new paradigm in computer networking. SDN makes computer networks programmable and more manageable by separating the control and forwarding components of the network where the centralized SDN controller programs and manages the network devices (e.g. switches, routers) by secured communication channel. The controller detects the link failure by receiving messages from the data plane and it can handle link failure both proactively or reactively. In the reactive approach, SDN controller installs alternative paths on demand in the

SDN switches which is time consuming. In the proactive approach, the controller installs redundant paths in the switches initially and the switches handle link failures locally by directing the flow to the alternative path.

In this project, based on SDN's proactive link failure handling approach we have come up with a better link failure handling mechanism for critical network infrastructures to ensure efficient path switching and restoration. We also compared the experimental results of our proposed mechanism with existing approaches for handling link failure. Our work includes the following:

- Calculation of the link failure delay of traditional network enabled with STP and RSTP.
- Implementation of the fast failover mechanism for SDN described in paper [3] and calculation of the delay to handle link failure.
- Development of an efficient failover recovery mechanism and calculation of link failure delay with it.
- Comparative analysis of the link failure delays from our experimental results.

From the comparative analysis of experimental results, we showed how our approach can provide a better solution to handle link failure in critical infrastructures. Finally, we have pointed out some limitations of our proposed failover recovery mechanism and presented possible solutions to overcome those limitations.

Chapter 2

Background Study and Related Works

2.1 Traditional Network and Approaches for Handling Link Failure

Traditional networks consist of hosts, switches and routers etc. where the network functionality and management are implemented in each of the network devices. The integration of control and forwarding behavior of the network in each of the devices makes the network time consuming and error prone. Traditional networks use spanning tree protocol and rapid spanning tree protocol for handling link failures where both of the protocols consume a significant amount of time in respect to critical network infrastructures.

2.1.1 Traditional Network Architecture

The traditional network architecture [4] consists of different types of devices e.g. switches and routers where the data plane and control plane are integrated into each of the devices. Each plane performs separate tasks that provide the routing functionality of the network. The control plane is responsible for configuration of forwarding devices and programming the forwarding behavior of the network. For example, in a traditional network switch, the spanning tree protocol is implemented and run in the network by the integrated control plane in the switches using different hello messages and bridge protocol data unit (BPDU) [5] packets. BPDU packets contain information on ports, addresses, priorities, costs etc and ensure that the data ends up where it was intended to go.

In a traditional network consisting of switches, the switches learn the MAC addresses of the devices in the network by using address resolution protocol (ARP) [6] messages. To prevent cycles in the network, the network devices intercommunicates and constructs a spanning tree in the network using spanning tree protocol [7] or rapid spanning tree protocol [8]. After learning the MAC addresses, the switches stores the MAC addresses and thus they become able to forward packets to their intended destinations. In the following figure, the topology is a traditional network where each switch has integrated control and data plane components inside them. Here, at (1) host $h1$ tries to send packets to $h2$, forwards it to switch

$s1$. At (2) switch $s1$ makes learns the MAC of $h2$ by broadcasting in the network, take forwarding decision and finally forwards the packet to $s2$. At (3) $s2$ forwards the packet to host $h2$. In this way, traditional network performs the forwarding functions.

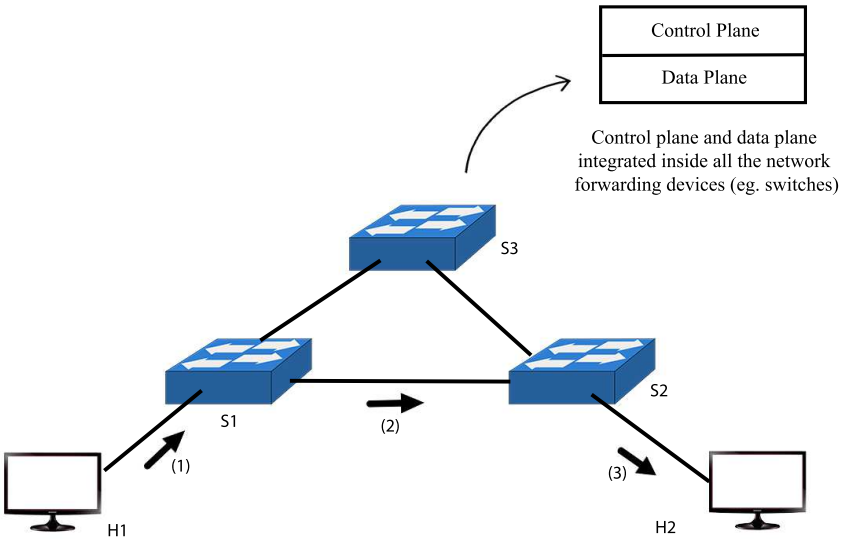

Figure 2.1: A traditional network where each Ethernet switch has integrated control and data plane inside.

However, the integration of control and data plane in traditional network devices has some drawbacks. If we want to change the behavior of the network, we need to reprogram all the network devices existing in the network. Because each device has control plane integrated inside them all of which must be programmed correctly for the network devices to run the network's forwarding functionality. As the control of the network is distributed in each device, the process of changing the network's behavior is not efficient. Another major drawback is the time taken for the network devices to initialize the forwarding behavior of the network is significant, as the network devices intercommunicate among themselves with their integrated control plane to control the behavior of the network. Thus, in unexpected situations like link failures, the traditional network fails to handle the link failure efficiently.

2.1.2 Spanning Tree Protocol (STP)

Spanning tree protocol (STP) [7] is used to prevent cycles in a traditional network topology. It also handles link failure in the network. It creates a spanning tree topology in the network by disabling the links which creates cycles. Spanning tree protocol prevents broadcast storm [9] created by broadcast packets in cyclic topology. In spanning tree protocol, network topology selects a switch as root or reference point. Root selection uses priorities and MAC addresses of the switches. Each switch has a priority associated with it. The default priority for the switch is 32768. Switch with the lowest priority is selected as root.

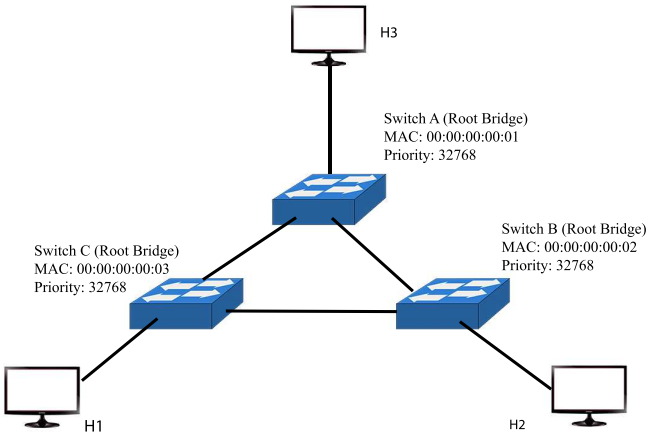

Figure 2.2: Root selection in STP

When priorities of all switches are same, the switch having lowest MAC address is chosen as root. Except for the root, each of the remaining switches selects a path for communicating with the root and blocks all other paths. This eventually connects all the switches of the topology without any redundant paths. If there are n numbers of switches in the topology, there are $n-1$ numbers of links connecting them.

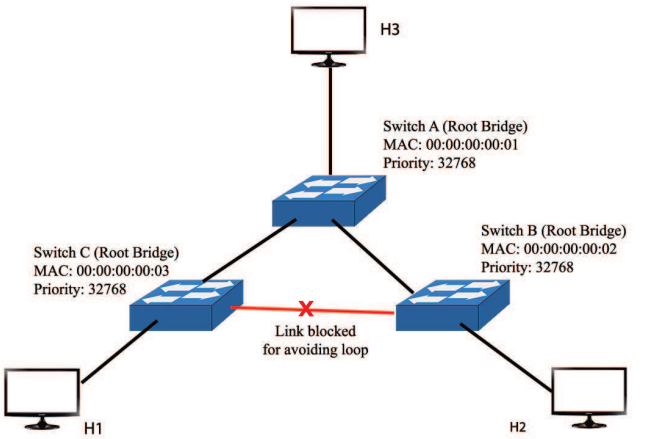

Figure 2.3: Blocking the link which creates cycle.

For blocking redundant paths and detecting link failure, switches need to share their information about themselves and their connections or links. For this purpose, small packets are exchanged among switches which are called BPDU (Bridge Protocol Data Unit) [5] packets. By default, the BPDU packets are exchanged in every two seconds. Each BPDU packet contains the following fields:

- Bridge ID: contains the sum of switch's priority and switch's MAC address. Root ID is unique for a switch because switch's MAC address is different from other switches.
- Root ID: contains the bridge ID of the root
- Root Path Cost: stores the path cost of a switch from the root

- Hello Timer: keeps the interval between the arrival of two consecutive BPDU packets
- Maximum Age: specifies the time to be taken before changing network topology

When a switch does not receive BPDU packets from the root for times specified in "Maximum Age" field (usually 20 seconds), switch considers topology has been changed and it needs to calculate a new path to the root. Then switch goes to the listening state. In listening state, switch unblocks a blocked path and exchanges BPDU packets for creating an active topology.

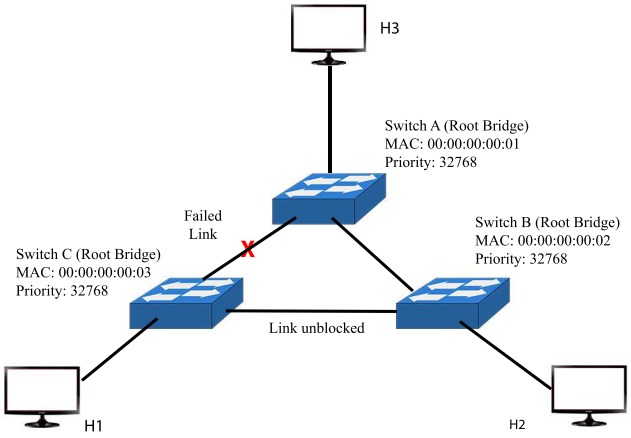

Figure 2.4: Unblocking the blocked link for handling link failure in STP

In listening state, switchs do not forward any data packets and consumes 15 seconds. After that, switches go to the learning state, where switch reads the MAC addresses of data packets it receives and places them in its MAC address tables as much as possible. Listening state also takes 15 seconds and does not forward any data traffic like the learning state. So, spanning tree protocol takes 30-50 seconds to respond to the change of topology. This time is called convergence time of spanning tree protocol. In this way, the spanning tree topology is created in the network using STP.

2.1.3 Rapid Spanning Tree Protocol (RSTP)

Rapid spanning tree protocol (RSTP) [8] is a modification of spanning tree protocol (STP) for which convergence time is significantly less than the convergence time of spanning tree protocol. RSTP enabled network labels four types of ports. Those are described bellow.

- Root port: A path from switch to the root. The best path among all paths to the root is selected as the root port. Best port means the closest or the fastest port.
- Designated port: A port that connects the switch with another switch. Designated port is non-blocking port that forwards traffic from one switch to another.

- Alternative port: A backup root port. When primary root port is failed, this port is used as root port.
- Backup Port: Alternative designated port, is kept to handle designated port failure

The purpose of keeping the alternative port and backup port is to re-converge as fast as possible after link failure. RSTP takes 6 seconds for detecting link failure where STP takes 20 seconds. As soon as RSTP detects link failure, the switch goes to the discarding state which replaces the disabled, blocking and listening states of STP. In discarding state, switch drops frames. Then switch goes to the learning state and then to the forwarding state. The absence of listening state saves 15 seconds. The learning state and forwarding state of RSTP work in the same way of those in STP. In RSTP, all switches including the root can exchange BPDU packets. So, in learning state, the switch which detects the link failure first, sends BPDU packets to the neighbors informing about the topology change. Then switches come to an agreement as soon as possible using the alternative or backup port. Then the switch detecting the link failure first goes to the forwarding state. The overall process takes less than 10 seconds.

RSTP can handle the link failure more efficiently than STP, however, in critical network infrastructures, the delay caused by RSTP to handle link failure is still significant. As the delay of even a few milliseconds can reduce the performance of critical networks, RSTP still fails to provide an efficient solution. So the link failure handling mechanisms in traditional networking cannot be a good option for critical network infrastructures.

2.2 Software Defined Network

Software defined networking (SDN) [10] is an approach to computer networking which separates the control of the network from it's forwarding behavior. SDN allows network administrators to control, change and manage the forwarding behavior of the network dynamically. It provides an application layer on the network by abstracting the lower level functionality of the network. By providing centralized control of the network, SDN overcomes different limitations of the traditional network system.

2.2.1 Software Defined Network Architecture

The architecture of software defined network can be divided into two separated components- the control plane and the data plane. The control plane consists of a centralized SDN controller which manages the network and controls it's forwarding behavior. The SDN controller takes the forwarding decisions in the network by programming the network's forwarding devices e.g. switches and routers through well defined communication protocol. Using communication protocols like OpenFlow [11] the physical infrastructure of the network is abstracted to the controller and the controller becomes able to program the network devices. The SDN controller also provides abstractions by creating an application layer on top the network to

provide APIs for the development of SDN applications and running them on the network. The data plane or the infrastructure layer consists of network devices e.g. switches and routers, along with the end hosts. The data plane only forwards the network packets instructed by the controller, it cannot take any forwarding decision. The communication between data and control plane is performed by OpenFlow messages through the secured channels connecting the switches to the controller.

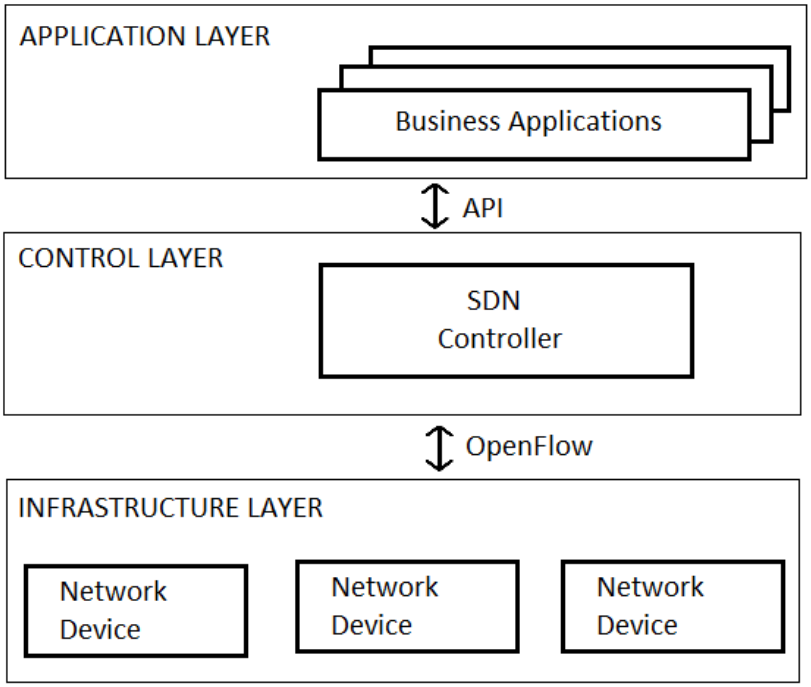

Figure 2.5: Architecture of Software Defined Network

As the SDN architecture is based on separation of control and data plane, with separated and centralized controller, the network's data plane becomes directly programmable and thus network administration becomes efficient and easier. The controller can provide abstraction to the application layer for developing different SDN application and running on top of the network. The centralized controller makes the network more secure by implementing different security mechanisms. Moreover, SDN simplifies network design and operation because abstraction of network's forwarding plane is provided to the controller using protocols like OpenFlow where any switch which supports OpenFlow protocol can be used in software defined networks and the internal composition of the switch is not a matter of concern for developing SDN applications.

2.2.2 OpenFlow Protocol

OpenFlow [12] is a communication protocol which defines standards for communication of data and control plane. Using OpenFlow the SDN controller can directly program the SDN enabled switches and thus control the network's forwarding behavior. The communication between controller and switch is done through secured

channels connecting the controller and each switch. A controller can control multiple switches and a switch can also be controlled by multiple controllers. At the time of initialization, the controller detects the network switches. This detection is done using different handshaking messages provided by OpenFlow. The switches periodically exchange LLDP (Link Layer Discovery Protocol) packets to each other to detect the existence of links to other switches. After detecting links, switches sends LLDP packets to the controller through the secured connection and then the controller learns about the topology of the data plane.

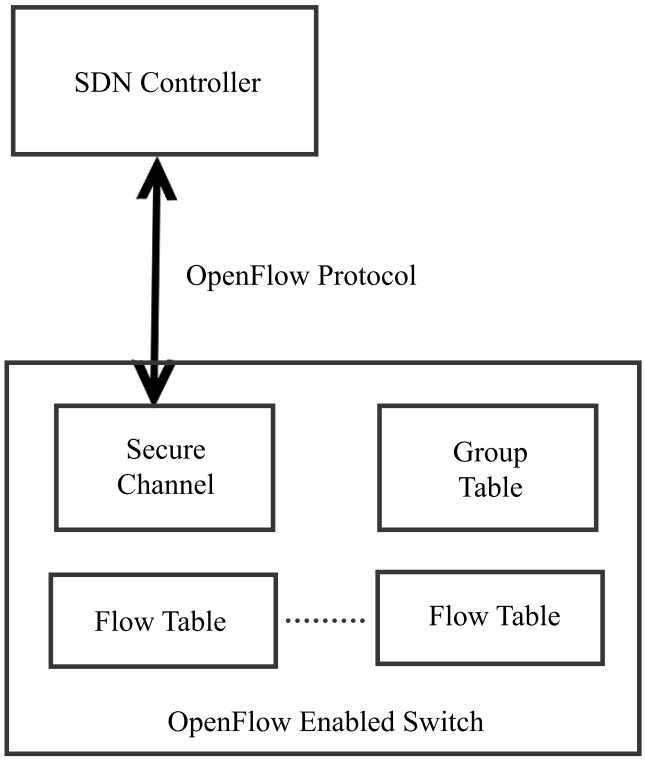

Figure 2.6: Communication of controller and switch with OpenFlow protocol.

To program the forwarding ability of the data plane, the OpenFlow protocol provides different types of messages like "Packet In", "Packet Out" and "Flow mod" etc. When a packet is received by a network switch which doesn't know how to forward the packet, it sends the packet to the controller through it's secure channel using the "Packet In" messages. Controller listens to the "Packet In" events and calculates the optimal path for the incoming packet. Then it installs the forwarding rules in the network switches according to the calculated path. The "Flow Mod" message are used by the controller to install forwarding rules in the switches, while "Packet Out" messages are used to instruct the switch simply to forward the packet. Thus, OpenFlow protocol can be used to program the network's data plane if the forwarding devices support it. For developing SDN applications, we do not need to get concerned about the internal composition of the forwarding devices, as any device which supports OpenFlow can be used to build software defined networks.

2.2.3 Flow Table

OpenFlow enabled switches contains flow tables [1, chapter 5.2] which store the forwarding rules and their corresponding actions. Each unit of rule and action is defined as flow table entry. When a packet enters in a switch, the packet headers are matched with the flow table entries and then the action associated with the matched entry is applied to the packet. The flow table entries contain the following fields:

- Match Fields: Packet headers of incoming packets are matched with the match fields.
- Priority: Flow entries of higher priority are preferred when a packet matches more than one flow entries.
- Instructions: Instructions are to modify the action set while pipeline processing.
- Timeouts: The maximum amount of time before a flow is expired in the switch.
- Cookie: A value which is set by controller.

SDN switches contain multiple flow tables arranged in a pipeline structure. The incoming packet enters the first flow table and in this way, the packet travels through the *Nth* table while the actions are added to the action set of the packet. After exiting the *Nth* flow table, the actions in the action set are applied to the packet. If no flow entries match the incoming packet, the situation is called table miss.

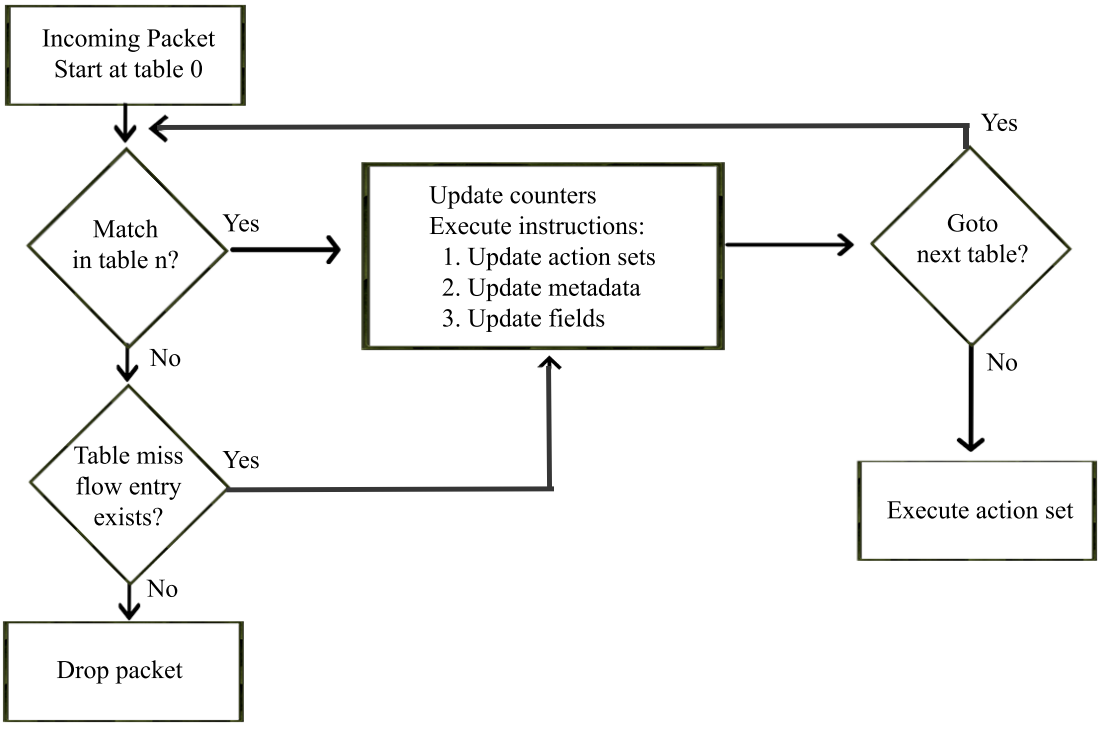

Figure 2.7: How packet is processed by flow table pipeline [1, figure 3]

The SDN controller can install flow entries in the switch using *FlowMod* messages. It can also remove the flow entries. There exist expiry mechanisms by which a flow entry is removed from the switch automatically after it expires.

2.2.4 Group Table

Switches which supports OpenFlow versions 1.1+ contains a special type of table called group table [1, chapter 5.6]. Group table contains group entries which contain a set of actions applied to the packets as action buckets. In the flow table entries, the actions can point to the group table entries to apply them on the matching packets. It's useful when more than one flow entries want to apply same actions to incoming packets, they can just point to the single group table entry which stores those actions in the action bucket. A single group entry contains the following items:

- Group Identifier: Uniquely identifies each group entry.
- Group Type: The type of the group entry which determines the group semantics.
- Counters: Stores the count of packets processed by the group entry.
- Action Buckets: Ordered list of action buckets, each containing a set of actions to apply to the incoming packets.

The *GroupType* field determines the way the action buckets are selected to apply to the incoming packets. Four different types of group entries are allowed in the group table, which are described in the following:

- ALL: All buckets in the group are executed.
- $SELECT$: One bucket from the bucket list is selected and executed.
- $INDIRECT$: Contains a single bucket which is executed.
- $FASTFAILOVER$: Executes the first live bucket.

Among the four types of group entries, the FAST FAILOVER type group entry is used in our experiments. To use this type of group entries, we have to set the group entry type to FF which means fast failover. Each action bucket in the fast failover group has a field called *watchport* which contains a port number. The liveness of that port defines the liveness of that bucket. If the port is down, the bucket will be non-live. In a fast failover group entry, the first live action bucket from the bucket list is executed. The structure of fast failover group entry is as follows.

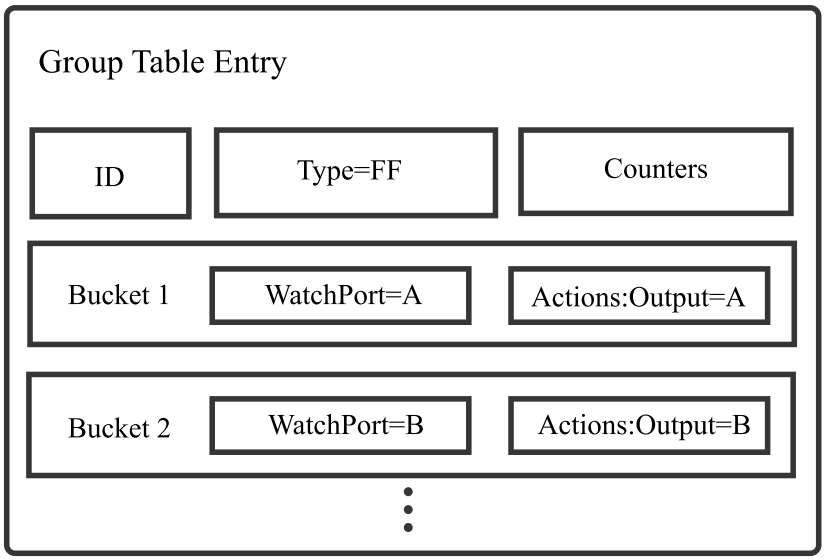

Figure 2.8: *FastFailover* type group table entry in a switch's group table

The advantage of FAST FAILOVER type of group entry is that the switch can handle link failure by changing the path of the data flow to next live action bucket in the group table entry. This change of path of the flow is instantaneous and takes a negligible amount of time. The use of this type of group entries can create scopes for the development of different fast failover mechanisms which can handle link failure more efficiently than the existing spanning tree protocols of the traditional network. So the software defined networking approach can be a better solution to handle link failure in critical network infrastructures.

2.3 Tools

We used different tools for conducting our experiments, creating the experimental network, implementing link failure handling mechanisms, observing the network's behavior and calculating the delay. For writing our scripts we used the API's provided by Mininet and to implement the failover mechanisms, we used the API provided by Ryu SDN Controller. We also used Wireshark, a packet analyzing tool to observe the path of the packets. Using these tools we have experimented and generated the results and thus we showed that our proposed failover recovery mechanism performs better than the existing mechanisms.

2.3.1 Mininet

Mininet [13] is a software emulator which can prototype a large network on a single computer. Mininet runs the network by creating a collection of hosts, switches, routers and links on a single Linux kernel. It uses lightweight virtualization to make a single system behave like a complete network. Each virtual host in a network created by Mininet behaves like a real machine and we can use it as an independent computer. We can write our own programs to run communication

between the hosts using the links available in the created network. This communication is facilitated by the standard communication protocols used for real internet. The links present in the network behave as the same way as Ethernet links. When two programs communicate through Mininet, the performance and behavior of the network can be used to evaluate how the network will perform in real life networks.

One important feature of Mininet is that it is very fast and takes almost a few seconds to create the network and make it functioning. Mininet is implemented in Python language which allows us to create custom topologies and experiment with them by providing well defined libraries. We can run our programs in the lightweight virtualized Mininet hosts and it will run the same way as it's running in the real life computer. The forwarding devices can be programmed using Mininet by means of SDN controller or direct manipulation of the behavior of network switches. For these reasons, Mininet is widely used for experimenting with network applications, protocols and security.

For software defined networking, Mininet provides support for running a network along with external SDN controller software. It also provides default controller application for simple SDN experiments. It allows the switches to connect the controller by TCP connection and allows OpenFlow communications between them. The command line interface (CLI) is an important part of the Mininet. It takes commands from the users at runtime and executes them on the network devices. Using CLI, we can modify the behavior of the network in many ways, for example, disabling a link, installing flow entries in flow tables of the switches, viewing the configuration of each node of the network, etc. Some basic uses of Mininet are described bellow.

We can create a simple topology consisting of one controller, one switch and two hosts using the command *sudomn* in the terminal. The following will be the output on the terminal.

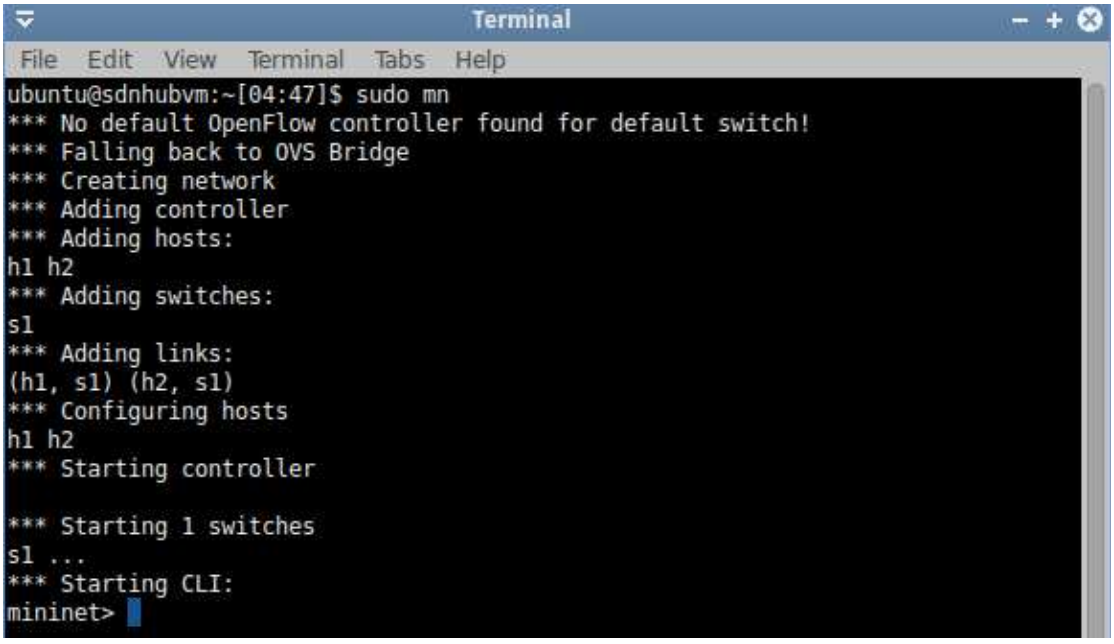

Figure 2.9: Creating a simple topology in mininet.

To view how many nodes and links are present in the network, we can use the commands *nodes* and *links*. The following figures indicates that the network consists of hosts $h1$,$h2$ and switch $s1$.

```
mininet> nodes
available nodes are:
h1 h2 s1
mininet>
```

Figure 2.10: List of nodes in the network.

The links present in the network are $h1 - eth0$ to $s1 - eth1$ and $h2 - eth0$ to $s1 - eth2$.

```
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
mininet>
```

Figure 2.11: Links in the network.

As each host behaves like a independent linux machine, we can run $ifconfig$ on it to see it's configuration.

```
mininet> h1 ifconfig
h1-eth0   Link encap:Ethernet  HWaddr b6:28:03:b0:fe:ac
          inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::b428:3ff:feb0:feac/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1206 (1.2 KB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

mininet>
```

Figure 2.12: Running $ifconfig$ on host $h1$.

The hosts can communicate with each other in the network. We can ping from one host to another.

```
mininet> h1 ping  h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.797 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.055 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.047 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.052 ms
```

Figure 2.13: Ping from $h1$ to $h2$.

The examples above are some basic uses of Mininet. Mininet also provides well defined and organized sets of API with which we can create and control a network. In our project, we have used different features of Mininet to create our experimental networks and test their performances.

2.3.2 Open vSwitch

Open vSwitch [14] is an open source virtual switch which supports communication through OpenFlow protocol [12]. They can be used to interconnect virtual machines within a host and virtual machines of different hosts connected by a network. It can be used for software defined networking switch as well as traditional network switch. It enables the Mininet to create a virtual network inside a single Linux based machine.

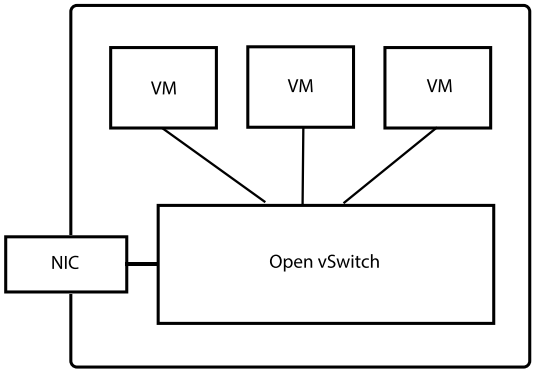

Figure 2.14: Open vSwitch interconnects virtual machines in a physical machine.

As Open vSwitch objects can be used to interconnect virtual machines in a host, it is used by Mininet to create virtual switches. As we mentioned above that Mininet used lightweight virtualized hosts in the network, Open vSwitch objects can be used to connect them. Mininet API provides a class called OVSSwitch which resembles the Open vSwitch. In our experiments, we used OVSSwitch objects to create the network switches in the experimental topologies.

2.3.3 Ryu Controller

Ryu controller is an open source software defined networking controller. The Ryu controller provides software components with well-defined application program interfaces to make it easy for developers to create new network management and controller applications. It helps organizations to customize the deployment of

their networks to meet their specific needs. The developers can quickly and easily create or modify existing components to ensure the underlying network can meet the changing demands of their applications. The Ryu Controller source code is hosted on GitHub [15] and managed by the open Ryu community. To support communication between control and data plane, Ryu controller supports OpenFlow protocol. It uses OpenFlow to interact with the forwarding plane (switches and routers) to modify how the network will handle traffic flows. It has been tested and certified to work with different OpenFlow-enabled switches, including Open vSwitch. In our experiments in chapter 7 and 8, we have used Ryu controller and developed applications on it which implements different fast failover mechanisms.

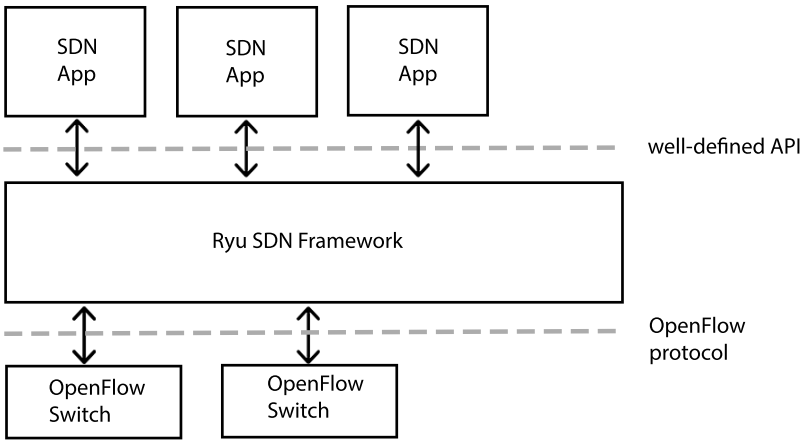

Figure 2.15: Architecture of Ryu SDN controller.

2.3.4 Wireshark

Wireshark [16] is a network packet analyzer. It can capture network packets from the available network interfaces in the machine and analyze them. It is a great tool which can be used in network administration, examining network security, learning the protocols etc. We used Wireshark to examine the flow of packets through different ports of the Open vSwitch switches and it was useful for finding the path of the network traffic in the experiments in next chapters.

2.4 Related Works

As the link failure handling with software defined networking is an emerging concept, different fast failover mechanisms has been proposed in different papers. We concentrated on four different types of fast failover mechanisms and done a comparative analysis of the complexities of them. We choose a simple and efficient fast failover mechanism proposed in the paper [3] and modified it with new and efficient features of software defined networking to make it more efficient and applicable for using in critical network infrastructures where a delay of few milliseconds can be a big concern.

One mechanism was proposed in the paper [2] for handling path failures and restoration involves controller on demand. The controller contains information about the complete topology. In case of any path or link failure, a switch informs the controller. When the controller is notified about a path or link change, controller checks each calculated path P whether it is affected by the link change or not. If any path is affected, the controller calculates a newly available path $P1$ for those end hosts. It also checks that if it has added flow entries in OpenFlow switches regarding the older faulty path. If so, then controller deletes the flow entries from all the OpenFlow switches regarding the older path and then adds flow entries in all the OpenFlow switches for the new path. One of the major drawbacks of this mechanism is that the centralized controller would be overloaded and would introduce scalability problems, and so isn't preferable for critical internet infrastructure.

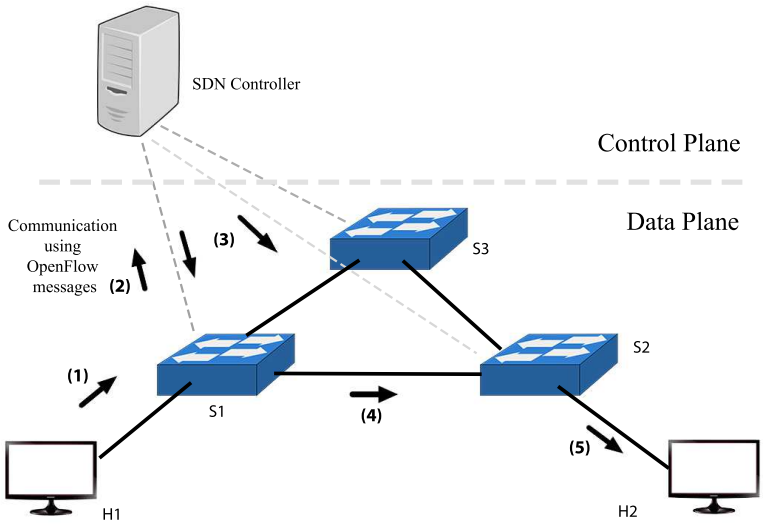

Figure 2.16: Controller installs path in the switches on demand as described in [2]

In above figure, At (1) host $h1$ tries to send packet to $h2$. At (2) $s1$ receives the packet and does not know how to forward it. So it forwards the packet to the controller in "packet in" message using OpenFlow protocol. At (3) Controller installs flow entries in switches $s1$ and $s2$ and at (4) switch $s1$ forwards the packet to $s2$ using the flow entries installed by controller. Finally at (5) $s2$ forwards the packet to $h2$.

The paper [17] described a proactive approach by which the controller periodically computes multiple paths for all source to destination pairs and installs the flow and group entries on the related switches. When a link failure is detected, the switch can failover the affected flows to the backup path. In this way, the connectivity of hosts could be recovered in a very short time. In the fast switchover mechanism, the controller periodically performs congestion detection for each port of each switch. When a port becomes congested, the controller adaptively decreases the transmission rate by iteratively switching the flow with the minimum rate to the backup path. Therefore, the link congestion could be

handled efficiently. An emulation using Mininet and Ryu controller demonstrates the fast switchover mechanism can reduce 47.5 to 72.5 percent sustained time of link congestion depending on the parameter setting. But controller overloading hasn't been reduced here.

According to the proposed mechanism of the paper [3], to protect against links and interface failure, the central OpenFlow controller (OFC) will compute the backup shortest path from source to destination based on the current topology information by excluding the main working path outgoing switch interface. Then, the controller will install the main and backup path information to respective OpenFlow separate switches flow table. If there is no failure happens during the transmission, the incoming packet will use the main path flow table to travel to the destination. When a failure disrupts the main working path, the affected OpenFlow switch will inform OpenFlow controller about the failure (the switch use Asynchronous messages to inform the controller of network events and changes).

While OpenFlow switch inform the controller about the failure, to avoid any delay that might interrupt the communication, the main path flow entry will be automatically evicted from switch flow table and it will switch to backup path to continue the traffic to destination (locally switch to provide a fast failover without a round trip to the central controller). To manipulate and develop a flow entry expiry mechanism that can reduce the network restoration, in this mechanism, the backup path configured will monitor the main path flow entry existence in the flow table. During the normal state, backup path flow will be in standby mode. When the main path link fail, the flow timeout of the entry will be expired automatically and once the entry is force to be deleted from the flow table, the backup path will be in active mode and continue to flow traffic thus guarantee minimal network restoration time (flow idle-timeout value will start to ensure flow table size is not wasted). The mechanism is implemented by us using the following algorithm:

Algorithm 1: Fast failover mechanism using flow table

```
foreach pair of hosts h1, h2 in the network do
    calculate main and backup path for src=h1,dst=h2;
    construct flow entries for main path with high priority;
    construct flow entries for main path with low priority;
    store the flow entries in the flow tables of the affected switches;
end
while running do
    if link s1-s2 fails then
        controller removes all flow entries from s1 and s2 which involved the
         failed link;
        s1 and s2 forward packets using the backup path;
        controller calculates and installs new main and backup path;
    else
        switches forward packets through the main path;
    end
end
```

The paper [18] proposed three different local fast failover mechanisms have

been proposed. Those mechanisms do not involve SDN controller's participation in handling link failure at real time. Instead, the proposed mechanisms use the packet tagging feature [1, Chapter 5.10] along with fast failover type groups and flow tables to deal with link failure locally. Packet tagging is a feature of OpenFlow which allows the OpenFlow enabled switches to attach and remove tags in the packet header space. The tags can contain information defined by the switches, so they were used to contain the parent node information in the topology and thus the proposed fast failover mechanisms make the packet travel through the network making a spanning tree when link failure happens. The mechanisms were evaluated with three type of complexities:

- Rule complexity: Total number of rules needed per switch.
- Tag complexity: Maximum tag space needed in a packet.
- Route length: Longest path traveled by the packet.

As our concern is to reach the packets to their destination as fast as possible, the rule and tag complexity are not taken into consideration in our work. The time complexity is the matter of concern in our work which is related to the route length of the algorithms. Three different local fast failover algorithms are mentioned in the paper: The Modulo Algorithm, Depth First Search and Breadth First Search. The modulo algorithm uses a round robin fashion to forward packets in alternative ports after the link failure. The ports are numbered from 1 to $2n - 1$ and when the port p_i fails, the switch will try to forward the packets through p_{i+1} and so on. In this way, the packet travels the network to reach its destination and in worst case, the packet has to travel far before reaching its destinations.

In the depth fast search (DFS) approach makes the packet travel in the network in a DFS fashion by constructing a spanning tree. The packet stores information of each switch and their parent switch using the packet tagging feature. Like the mod algorithm, the DFS algorithm tries to forward the packet to each neighbor switches. But if the switch fails to forward the packet to the neighbors, the packet is returned to the parent switch. This algorithm improves significantly over the mod algorithm and limits the path complexity to $O(n)$. The breadth fast search (BFS) is slightly different from the DFS approach as it creates a dependency between the path of the packet and the shortest route from the destination. In each step, the BFS algorithm searches the destination in the network by increasing d, the destination radius. If destination is not found in d radius, the BFS algorithm expands the search area by d and it can perform better than the DFS approach of making the packet travel the whole network topology.

The mechanisms for handling link failures which are described above, handles link failures in different ways. The paper [2] describes a reactive approach where controller installs alternative paths on demand, which is time consuming. The paper [2] describes a congestion aware failover mechanism which also protects congestion. Three different kinds of graph algorithms have been used in paper [18] which handles the link failure locally and without interrupting the controller. In our project, we focused on the paper [3] which proposes a redundant path approach to handle link failure using flow table entries. It was modified in this project using group table to propose a more efficient fast failover mechanism which can be used in critical network infrastructures.

Chapter 3

Proposal: A Software Defined Networking Approach for Link Failure Recovery in Critical Network Infrastructures

Link failure is a common phenomenon in computer networks and it should be handled efficiently to uphold the performance of the network. In this project, we have proposed a failover recovery mechanism for critical network infrastructures. In computer networking, failover recovery mechanism is defined as the process of instant switching of the network traffic flow to the redundant path in case of link failure. Our proposed mechanism is a modification of an existing mechanism described in paper [3].

Existing fast failover mechanism and it's limitations

The paper [3] proposes a fast failover mechanism which handles link failures in software defined networks by installing main and backup paths in the network after detecting the link failure. According to this mechanism,

- The controller installs two main and backup paths in the network using flow table entries.
- When main path fails, the controller removes the flow entries of the main path and the network traffic is directed through the backup path
- The controller installs new main and backup paths in the network and the old paths are removed eventually using flow expiry mechanism.

However, one limitation of the fast failover mechanism proposed in [3] is that when the main path fails, the controller will have to remove the path from the network. Until then the flow of packets will stop because the switches will keep trying to forward packets through the failed link. After the flow entries of the failed link are removed, the flow of traffic in the network will be restored. If the communication of the controller and switch are delayed, the controller will take significant amount of time to remove the flow entries of the failed link and hence, the flow of packets will be stopped temporarily and thus the performance of the network will be degraded.

Our proposed failover recovery mechanism

Our proposed failover recovery mechanism overcomes the limitations described in the previous section and improves the performance significantly. The fast failover mechanism of the paper [3] used flow table entries for storing main and backup paths. Instead of using only flow entries, we have used both flow and group table entries in our proposed mechanism. We have used fast failover type group table entries [19] for storing main and backup path together in single group entry. The controller sets the main and backup paths as first and second action buckets respectively. In fast failover group, the first live bucket is used to forward packets. So, packets will always be forwarded with the main path and in case of link failure, they will use backup path. The algorithm of the proposed failover recovery mechanism is stated bellow:

Algorithm 2: Proposed failover recovery mechanism using group entries of fast failover type.

```
foreach pair of hosts h1, h2 in the network do
    Controller calculates main and backup path for src=h1,dst=h2;
    Controller constructs fast failover group entries where 1stbucket=main
     path and 2ndbucket=backup path;
    Controller stores the group entries in the group tables of the affected
     switches;
    Controller stores the flow table entries which points to the group entries
     mentioned in above step;
end
while running do
    if link s1-s2 fails then
        Switches s1 and s2 forward packets through next live bucket in the
         group entry (backup path);
        Controller calculates and installs new main and backup path;
        Controller replaces the group entries of old main and backup paths
         with new ones.
    else
        Switches forward packets through the main path by sending packets
         according to first live bucket of the group entries;
    end
end
```

The following diagrams show how the proposed fast failover mechanism handles link failure by installing main and backup paths by group table entries. Let's consider the following topology containing switches $s1$,$s2$ and $s3$ and hosts $h1$and $h2$. According to our proposed algorithm, the controller calculates main and backup paths for hosts $h1$ to $h2$. The group table entry of the switch $s1$ will be fast failover type and it will contain the following action buckets.

- Bucket 1: if port $eth2$ is live, forward through $eth2$
- Bucket 2: if port $eth3$ is live, forward through $eth3$

The packets from $h1$ to $h2$ will be forwarded through the first live bucket which is the main path. The main path is shown in following figure.

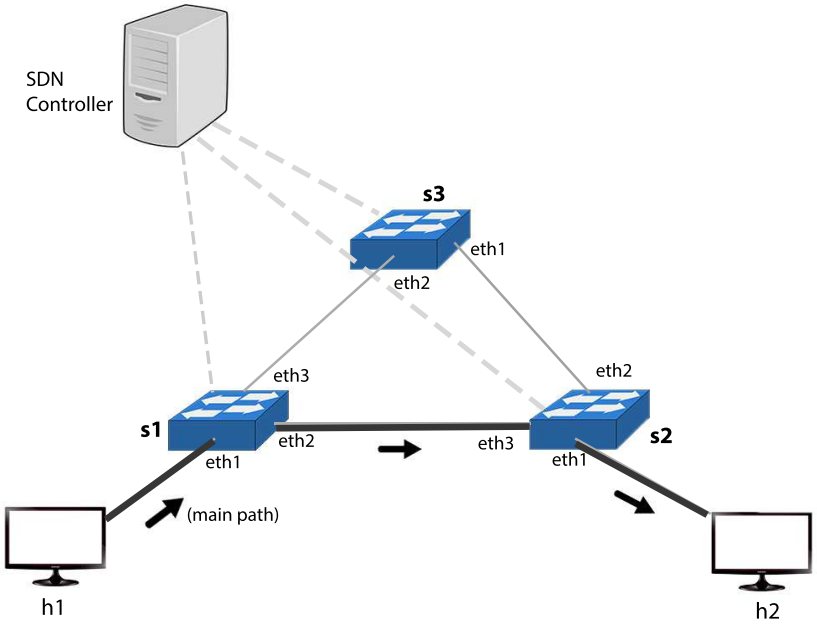

Figure 3.1: After controller installed main and backup path, the packets from $h1$ to $h2$ takes the main path.

When the port $eth2$ in switch $s1$ is down, the link between s1 and s2 fails. The bucket 1 will not be live and the switch will forward the packets to the next live bucket, which is bucket 2. The bucket 2 is the backup path. The following will be the path of the packets after the link failure.

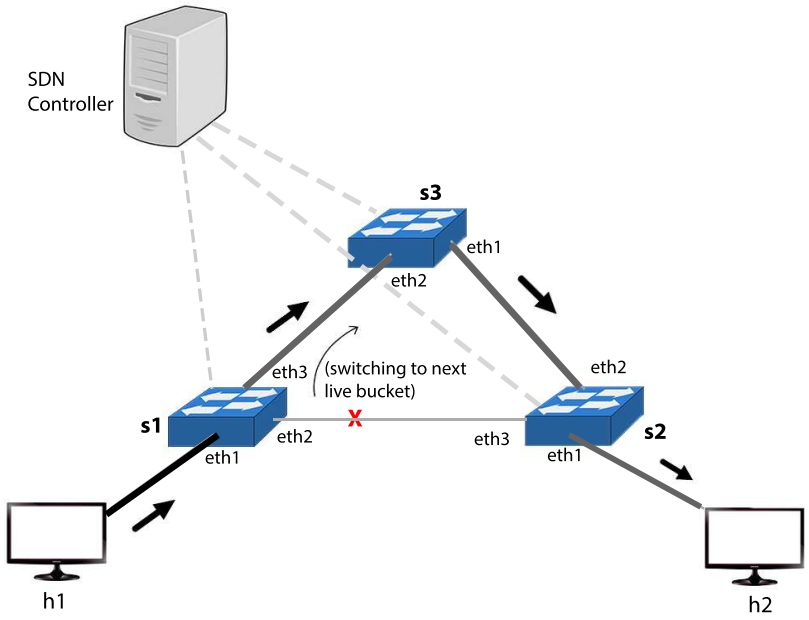

Figure 3.2: After link failure, the packets takes backup path.

In the above figure, the packets are switched to the backup path instantly after the link failure because after the link failure the switching from first non-live bucket to second live bucket happens locally and instantly in the switch. It takes a negligible amount of time and enhances the performance of our algorithm. In the following chapters, we have shown experimental results of different existing failover mechanisms and our proposed mechanism and compared the results. We have found that our proposed failover recovery mechanism performs better than the existing mechanisms. According to our experimental results, we have concluded that the proposed recovery mechanism can provide a better solution for handling link failure in critical network infrastructures.

Chapter 4

Experiments with link failure handling using existing mechanisms

4.1 Experiment 1: Link failure delay with Spanning Tree Protocol

Spanning tree protocol (STP) [7] prevents loop formation in the traditional networks and handles link failures. Our objective in this experiment is to calculate the delay to handle link failure in the traditional network using STP and to show that STP with traditional network cannot be an efficient approach for handling link failures in critical network infrastructures.

Description

We created a traditional network topology in Mininet[13] which consisted of a number of switches and hosts. The topology contains cycles in it. The switches used in the topology are Open vSwitch [20] objects. They can be used as traditional switches which learn MAC addresses in the network to forward packets. We enabled spanning tree protocol in the experimental network topology using the commands [21] provided by Open vSwitch. Finally, at the runtime we initiated a link failure in the network by disabling a link and calculated the time to handle the link failure by the spanning tree protocol. The scripts which were used to run this experiment can be found in [22].

Tools and Equipment

- Mininet: A software emulator for experimenting with large network in a single machine.
- Open vSwitch: A software implementation of virtual switch which behaves like real Ethernet switch.
- Wireshark: A packet capturing and analyzing tool.

Experimental Procedure

We conducted the following steps for this experiment.

Creating the network topology with spanning tree protocol on

We developed our experimental network topology by writing a script [23] using the API's provided by Mininet. The topology consists of six Open vSwitch objects named from $s1$ to $s7$ and two hosts $h1$ and $h2$. The links present in the topology are bidirectional and there exist cycles in the topology. The IP addresses of hosts $h1$ and $h2$ are 10.0.0.1 and 10.0.0.2 respectively. Using the API's provided by Mininet, we saved the ARP information of the hosts in each of the hosts. We set the switches in standalone mode which allowed the switch to behave like the MAC-learning switches of the traditional network. Then we enabled spanning tree protocol in each of the switches. The topology is as the following.

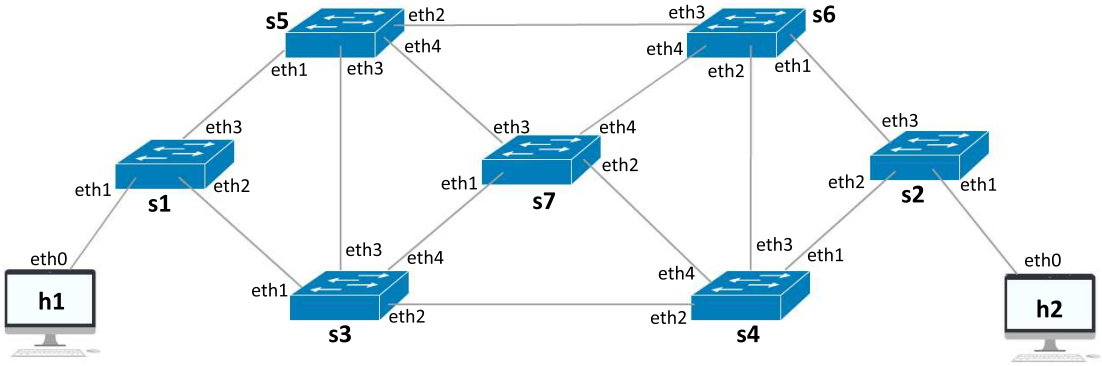

Figure 4.1: Traditional network topology for experimenting with spanning tree protocol (STP)

The above figure is the topology created at this step. The switches in the topology are traditional network switches where data and control plane are integrated. The switches started communicating with each other to establish the spanning tree protocol.

Running the communication between hosts in the network

In this step we ran one of our script [23] to conduct communication between host $h1$ and $h2$ in the experimental network. We executed a script to make the network run on the Mininet terminal. When the network started running, we made the host $h1$ send ping packets to host $h2$ continuously and wait for ping reply to come back to $h1$. The spanning tree protocol took some time to converge with the network and then host $h1$ could make a successful ping. We calculated the time taken by spanning tree protocol to converge by calculating the time difference between the first successful ping and the time when the network started functioning. After spanning tree protocol was finally converged the following was the output of the Mininet terminal.

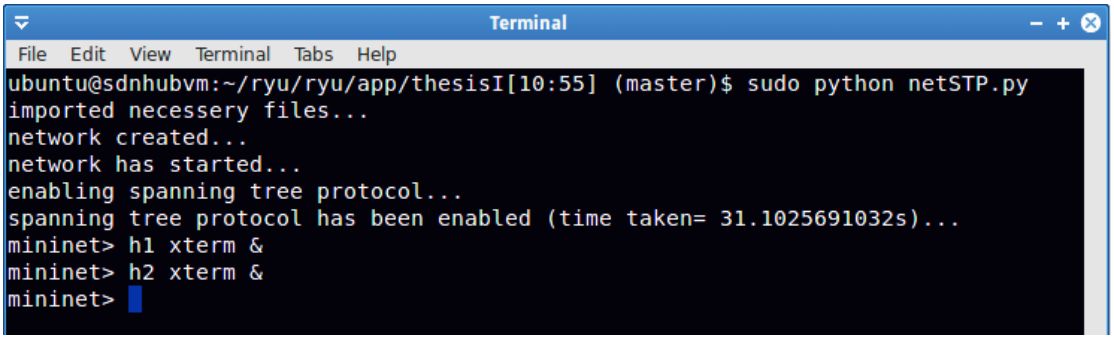

Figure 4.2: Mininet terminal after the spanning tree protocol converged with the network

We opened two xterm [24] windows for host $h1$ and host $h2$. Xterm allows us to run applications inside any virtual host created by Mininet. In the xterm of host $h1$ we ran a script [25] to make the host $h1$ send continuous UDP packets to host $h2$. The data inside the UDP packet contained only the sequence number of the packet generated at $h1$. After sending each packet, $h1$ showed output on the xterm which indicated a packet was sent.

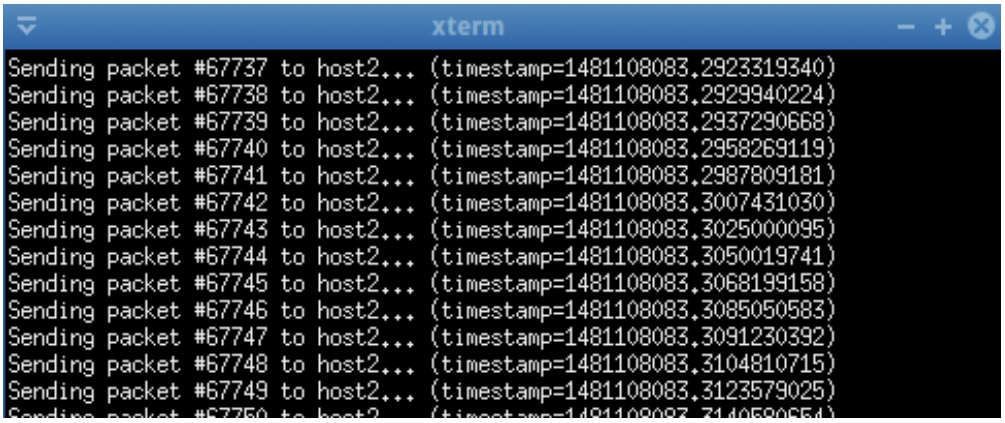

Figure 4.3: Host $h1$ sending packets to host $h2$

In the xterm of host $h2$, we ran a script [26] it received the UDP packets sent by host $h1$. The packets traveled throw the network and reached host $h2$. After receiving an UDP packet, host $h2$ calculated the time interval of the current packet with the previously received packet. The host $h2$ also calculated the average time. Then the host $h2$ printed a log in its xterm showing the sequence number from the packet, the timestamp of arrival, the time interval of the packet with the previous packet and the average time of arrival. The host $h2$ also writes these information on a log file [27].

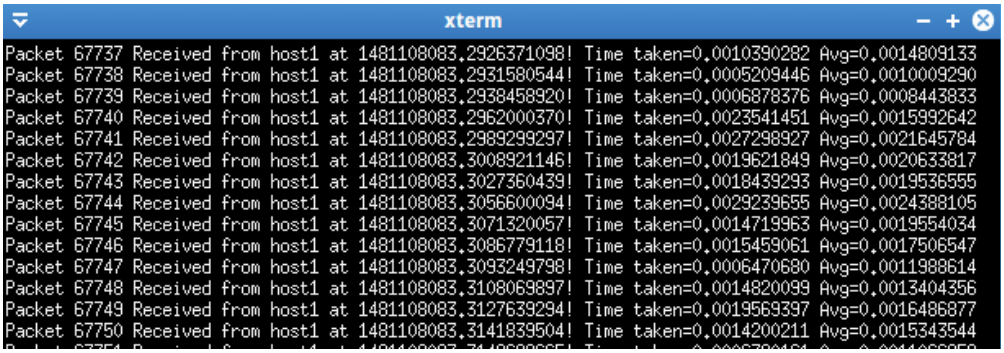

Figure 4.4: Host $h2$ receiving packets from host $h1$

We observed the flow of packet on each port of each switch using Wireshark. We ran Wireshark and opened the capture interface. The capture interface contains the list of ports on each switch and the counts of total packets forwarded through the port. The output of the capture interface was as the following:

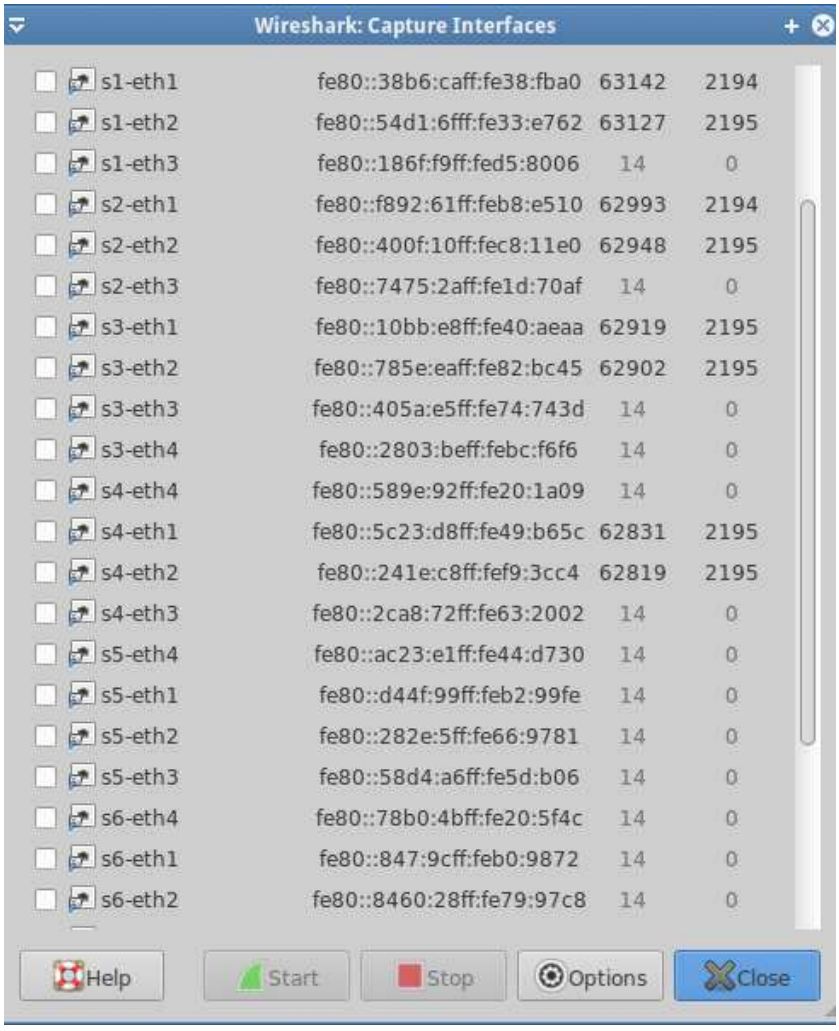

Figure 4.5: Wireshark capture interface showing counts of captured packets through each port of each switch

The figure above is an Wireshark window showing the packet counts of each

port of each of the switches in our experimental topology. Here some ports have packet counts significantly higher than the others. Those port are forwarding the UDP packets from $h1$ to $h2$. The path of the packet from host $h1$ to host $h2$ as $h1 \rightarrow s1 \rightarrow s3 \rightarrow s4 \rightarrow s2 \rightarrow h2$ which is shown in the following figure:

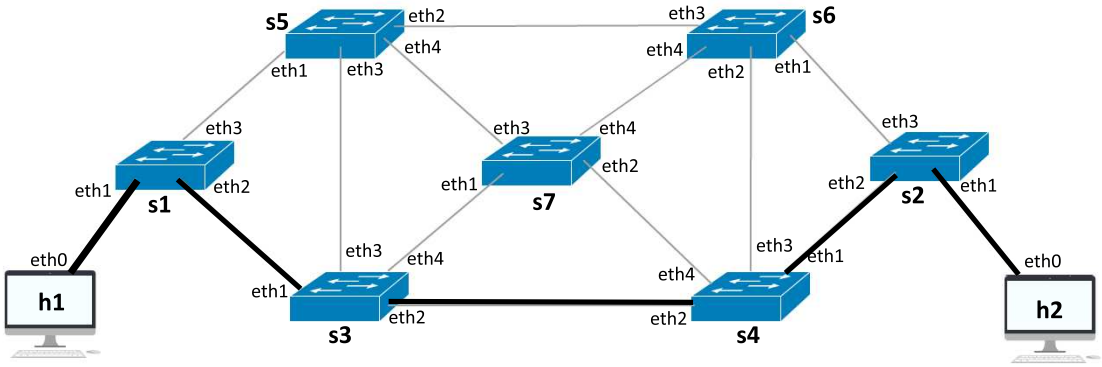

Figure 4.6: Path of the packets in the network

Thus, we determined the path taken by the experimental UDP packets from $h1$ to $h2$ in our network topology enabled with spanning tree protocol.

Initiating link failure in the network

In this step, we ran a script [28] to initiate a link failure in the network while host $h1$ and $h2$ are communicating with each other. We opened xterm windows for $h1$ and $h2$. Then we made $h1$ send UDP packets to $h2$ by running scripts in these hosts. For testing link failure, we ran a script which disabled the port $s3-eth2$ in the switch $s3$. The behavior of the switches can be controlled in Mininet through different commands for Open vSwitches. After disabling the link, the timestamp of the link failure was shown on the terminal. From the terminal we see that the timestamp when link failure happened is 1481108516.67482090.

```
ubuntu@sdnhubvm:~/ryu/ryu/app/thesisI[03:01] (master)$ sudo python linkbreaker.p
y
disabling port s3-eth2
Port s3-eth2 was disabled at 1481108516.6748209000
ubuntu@sdnhubvm:~/ryu/ryu/app/thesisI[03:01] (master)$
```

Figure 4.7: Initiation of the link failure in the network

Observation

In this step, we observed the backup path of the packets going from $h1$ to $h2$ after the link failure. In the xterm of host $h1$, we observed that after initiation of link failure, the host $h1$ was still sending UDP packets to the host $h2$. However, in the xterm of $h2$, we observed that just after initiation of link failure, the host $h2$ stopped receiving packets from host $h1$. After some time, the host $h2$ started receiving packets again. This delay happened because the spanning tree protocol

consumed some time to re-converge in the network after the link failure. We opened the Wireshark's capture interface showing packet counts of each port to determine the path of the packets after the link failure.

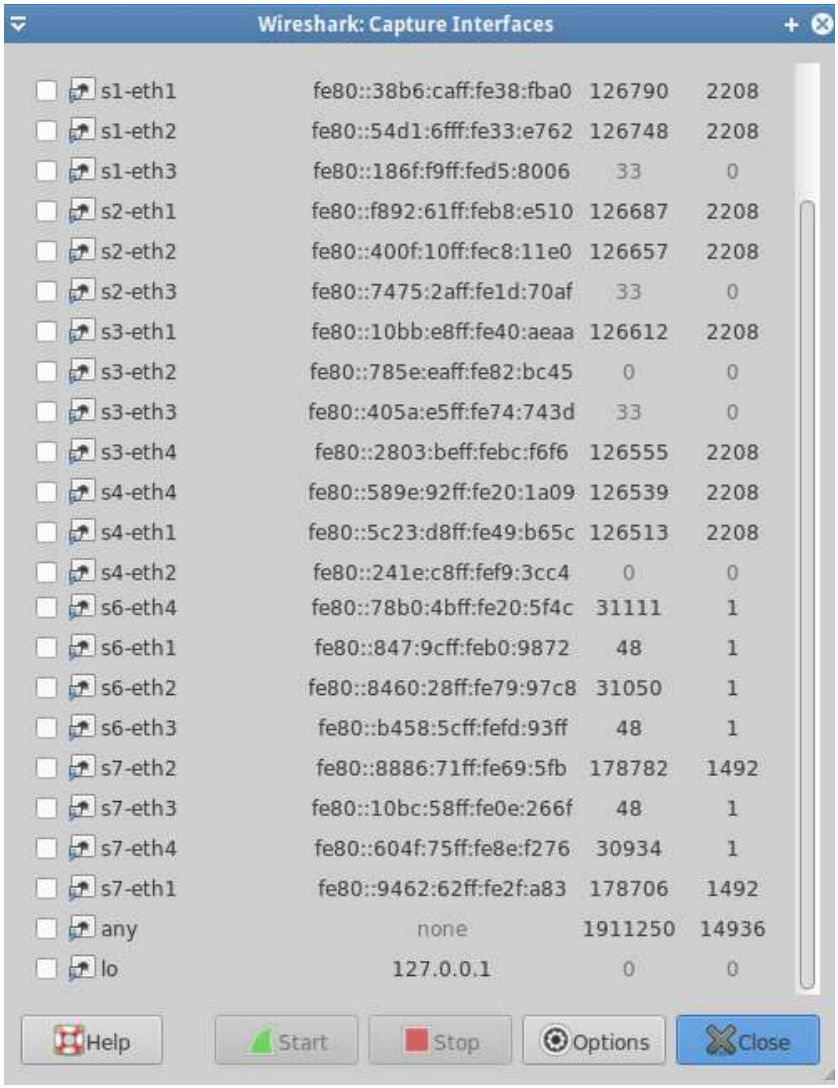

Figure 4.8: Wireshark's capture interface showing the counts of packets through each port of each switch after link failure

From the observed data from Wireshark's capture interface on above figure, we determined that after spanning tree protocol re-converged with the network after link failure, the path of the packet from host $h1$ to host $h2$ as $h1 \rightarrow s1 \rightarrow s3 \rightarrow s7 \rightarrow s4 \rightarrow s2 \rightarrow h2$ which is showed in the following figure:

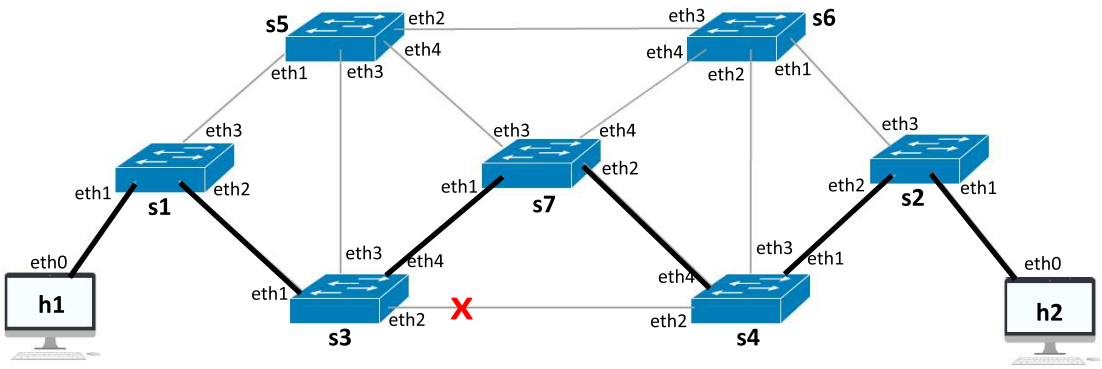

Figure 4.9: Path of the packets after the spanning tree protocol handled link failure

Calculation of the time to handle link failure

After causing link failure in the network, we calculated the time to handle link failure in this step. We know that the timestamp of the link failure T_f=1481108516.6748209. Earlier we said that the host $h2$ keeps a log file of the received packets. We observed that log file and looked for the first packet which was received after T_f. The following section of the log file [27] contains that packet.

```
Packet 24855 Received from host1 at 1481108516.6524341106! Time taken=0.0086131096 Avg=0.0057604079
Packet 24856 Received from host1 at 1481108516.6547100544! Time taken=0.0022759438 Avg=0.0040181758
Packet 412392 Received from host1 at 1481108566.2600469589! Time taken=49.6053369045 Avg=24.8046775
Packet 412504 Received from host1 at 1481108566.2606289387! Time taken=0.0005819798 Avg=12.40262976
Packet 412505 Received from host1 at 1481108566.2610459328! Time taken=0.0004169941 Avg=6.201523377
```

Figure 4.10: Log file containing information about packets received by $h2$

From the above figure, we determined that the packet of index 412392 was received first after the time T_f. The packets between 24856 and 412392 were not received by host $h2$ because of the link failure. So, the time interval associated with that packet is the delay caused by spanning tree protocol to handle the link failure, which is 49.6053369 seconds.

Summary of the experiment

From the experimental result, we found that the time to handle link failure by spanning tree protocol is significantly high because spanning tree protocol takes a lot of time to get converged in the network. So, we can conclude that spanning tree protocol (STP) is not suitable for using in critical network infrastructure.

4.2 Experiment 2: Link failure delay with Rapid Spanning Tree Protocol

The objective of this experiment is to calculate the delay to handle link failure in the traditional network using rapid spanning tree protocol (RSTP) [8]. Rapid spanning tree protocol is a modification of spanning tree protocol which can handle link failure more efficiently than spanning tree protocol. We showed that RSTP improves the performance over STP by minimizing the link failure delay but the delay can be significant with respect to critical network infrastructures.

Description

We created a traditional network topology using Mininet [13]. The network topology consists of traditional mac-learning virtual switches and hosts. There are cycles present in the network. The switches used in the topology are Open vSwitch [20] objects. We enabled rapid spanning tree protocol in the network using the commands provided by Open vSwitch. After the network started running, we initiated a link failure in the network by disabling a link and then calculated the time to handle the link failure by the rapid spanning tree protocol. The scripts which were used to run this experiment can be found in [29].

Tools and Equipment

- Mininet: A software emulator for experimenting large network in single machines
- Open vSwitch: A software implementation of virtual switch which behaves like real Ethernet switch.
- Wireshark: A packet analyzing tool.

Experimental Procedure

For experimenting with rapid spanning tree protocol, we conducted the following steps.

Creating the network topology with Rapid Spanning Tree Protocol on

We developed our experimental topology using the API's provided by Mininet. The topology consists of six switches named from $s1$ to $s7$ and two hosts $h1$ and $h2$. The links present in the topology are bidirectional and there exist cycles in the topology. The IP addresses of hosts $h1$ and $h2$ are 10.0.0.1 and 10.0.0.2 respectively. The ARP information of each host is stored in all of the hosts present. We set the switches in standalone mode which allows the switch to behave like the MAC-learning switches in the traditional network. Then we enabled rapid spanning tree protocol in each of the switches. The topology is as follows.

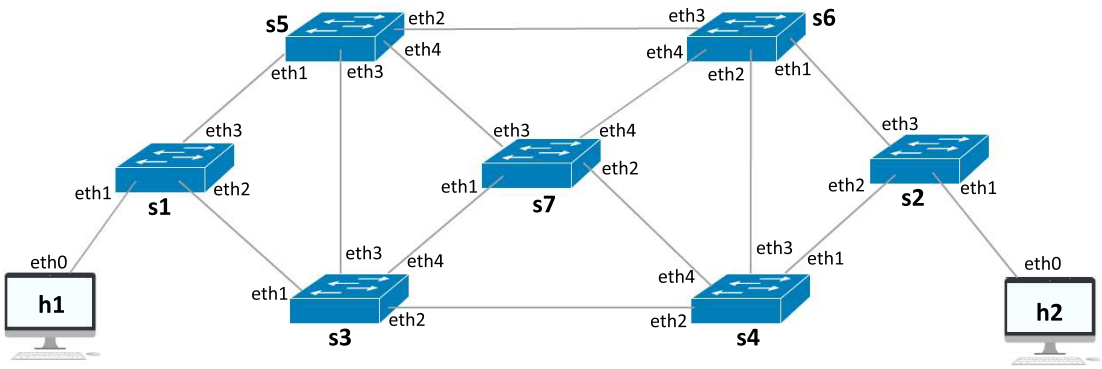

Figure 4.11: Traditional network topology for experimenting with rapid spanning tree protocol (RSTP)

The figure above is the topology created by the end of this step. The topology is a traditional network topology enabled with rapid spanning tree protocol.

Running the communication between hosts in the network

In this step we conducted communication between host $h1$ and $h2$. We started the network and then we made the host $h1$ send ping packets to host $h2$ and wait for ping reply. The rapid spanning tree protocol took some time to converge with the network and then host $h1$ could execute a successful ping. We calculated the time taken by rapid spanning tree protocol to converge by calculating the time difference between the first successful ping and the time when the network started functioning. After the rapid spanning tree protocol was finally converged, the following was the output of the Mininet terminal.

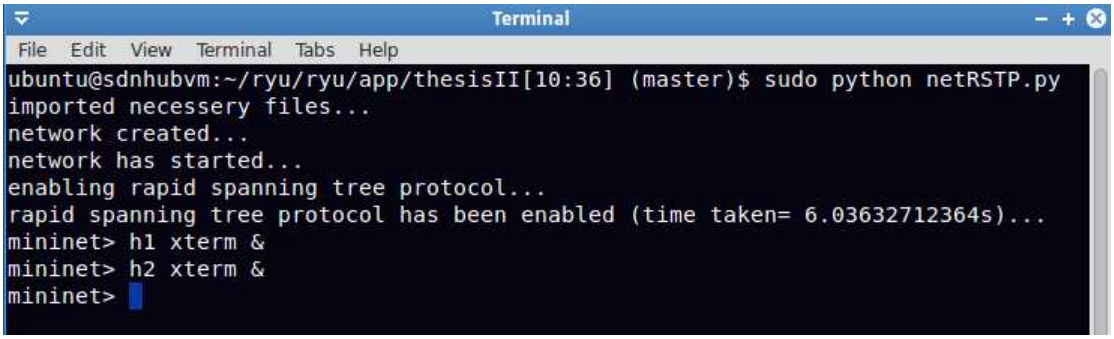

Figure 4.12: Mininet terminal after the rapid spanning tree protocol converged with the network

We opened two xterm [24] windows for host $h1$ and host $h2$ respectively. In the xterm window of host $h1$, we ran a script which made the host $h1$ send continuous UDP packets to host $h2$. The data inside UDP packet contains only the sequence number of the packet. After sending each packet, $h1$ showed output on the xterm window.

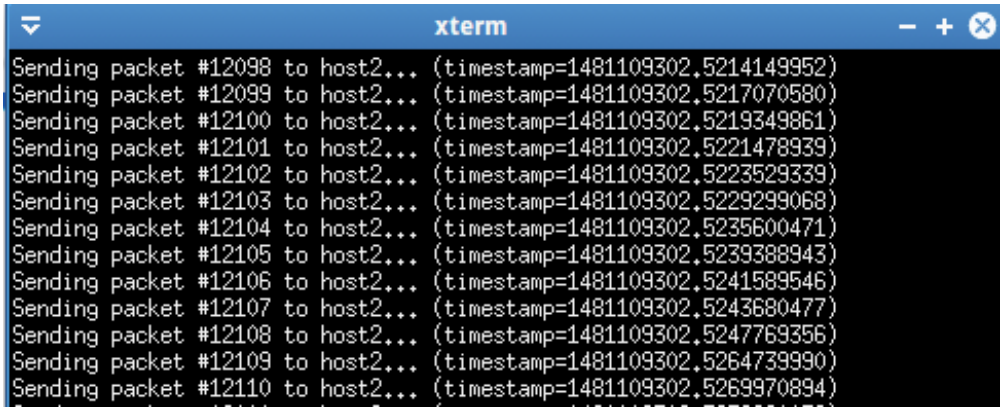

Figure 4.13: Host $h1$ sending packets to host $h2$

In the xterm of host $h2$, we ran a script to make the host $h2$ receive the UDP packets sent by host $h1$. The packets traveled throw the network and reached host $h2$. After receiving an UDP packet, host $h2$ calculated the time interval of the current packet with the previously received packet. The host $h2$ also calculated the average time. Then the host $h2$ printed the sequence number from the packet, the timestamp of the packet, the time interval of the packet with the previous packet and the average time on the xterm. The host $h2$ also writes this information in a log file.

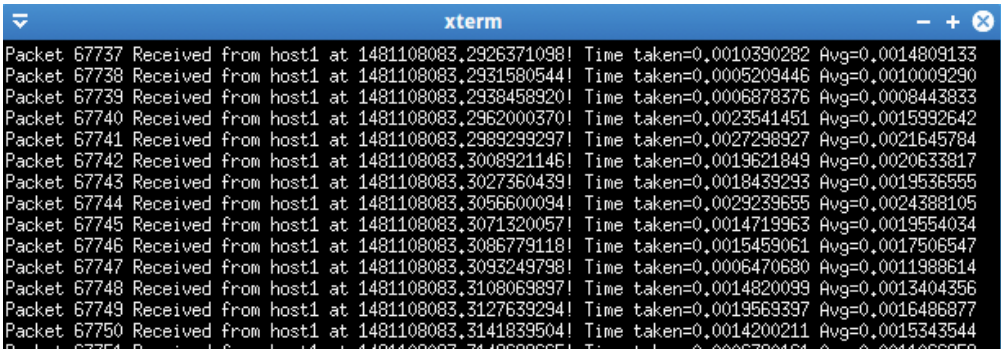

Figure 4.14: Host $h2$ receiving packets from host $h1$ and showing output on the terminal

We observed the traffic on each port of each switch using Wireshark. We ran Wireshark and opened the capture interface. The capture interface contains the list of ports on each switch and the counts of total packets forwarded through the port. The output of the capture interface was as the following:

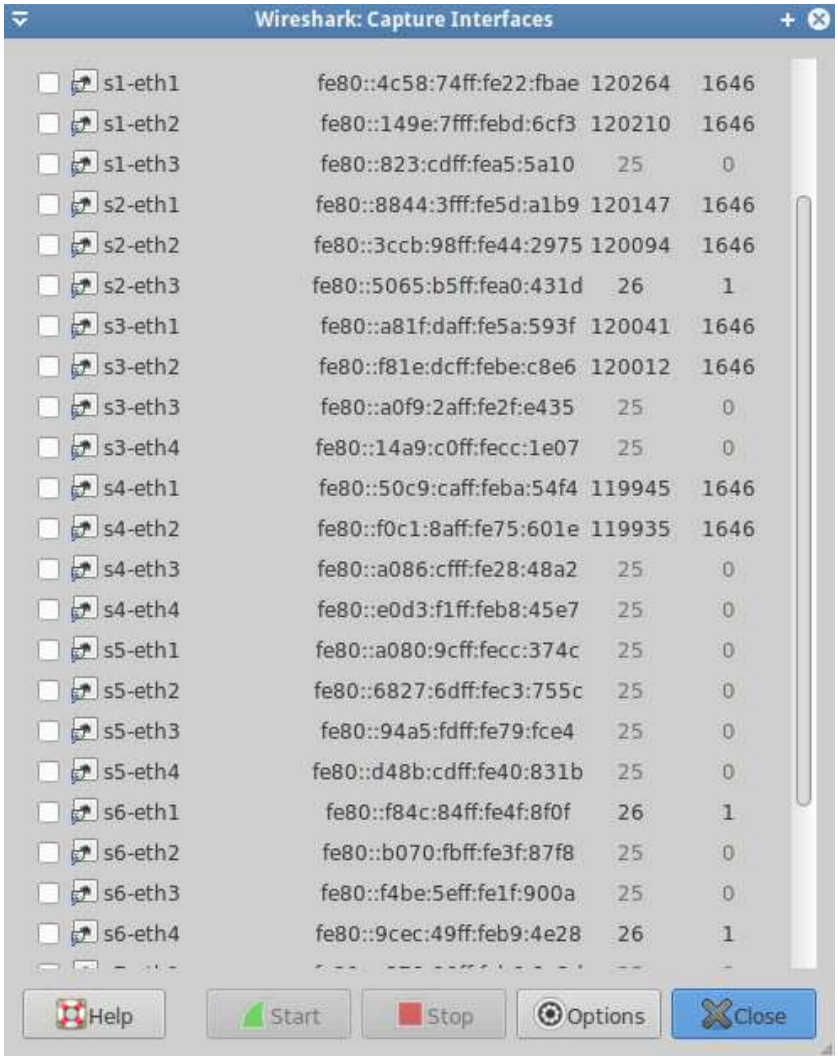

Figure 4.15: Wireshark capture interface showing packet count of each port of each switch

The above figure is a Wireshark window showing the packet counts of each port of each of the switches in our experimental topology. Here some ports have packet counts significantly higher than the others. Those port are forwarding the UDP packets from $h1$ to $h2$. From the above data from Wireshark, we found the path of the packet from host $h1$ to host $h2$ as $h1 \rightarrow s1 \rightarrow s3 \rightarrow s4 \rightarrow s2 \rightarrow h2$ which is showed in the following figure:

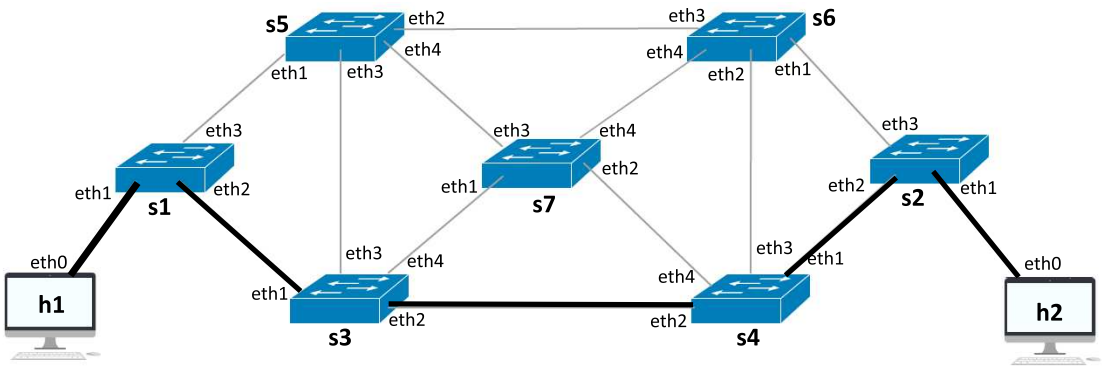

Figure 4.16: Path of the packets in the network

Thus, we determined the path taken by the experimental UDP packets from $h1$ to $h2$ in our network topology enabled with spanning tree protocol.

Initiating link failure in the network

In this step, we initiated a link failure in the network while $h1$ sending continuous UDP packets to $h2$. We opened xterm windows for $h1$ and $h2$ respectively and made host $h1$ send UDP packets to host $h2$. For testing link failure, we ran another script which disabled the port $s3 - eth2$ in the switch $s3$. The behavior of the switches can be controlled externally in Mininet through different commands for Open vSwitches. After disabling the link, the timestamp of the link failure was shown on the terminal. From the terminal, we see that the timestamp when link failure happened is 1481109655.3601229.

```
ubuntu@sdnhubvm:~/ryu/ryu/app/thesisII[03:20] (master)$ sudo python linkbreaker.
py
disabling port s3-eth2
Port s3-eth2 was disabled at 1481109655.3601229191
ubuntu@sdnhubvm:~/ryu/ryu/app/thesisII[03:20] (master)$
```

Figure 4.17: Initiation of link failure.

Observation

In this step, we observed the effect of the link failure on the network and the backup path of packets from $h1$ to $h2$. In the xterm of host $h1$, we observed that after initiation of link failure, the host $h1$ was still sending UDP packets to the host $h2$. However, in the xterm of $h2$, we observed that the host $h2$ stopped receiving packets for a short moment and then again started to receive packets. The time interval was very short here as the rapid spanning tree protocol is faster than spanning tree protocol. We opened the Wireshark's capture interface to determine the path of the packets after the link failure. From the observed data from Wireshark's capture interface, we found that after rapid spanning tree protocol re-converged with the network after link failure, the path of the packet from host $h1$ to host $h2$ as $h1 \rightarrow s1 \rightarrow s5 \rightarrow s6 \rightarrow s2 \rightarrow h2$ which is showed in the following figure:

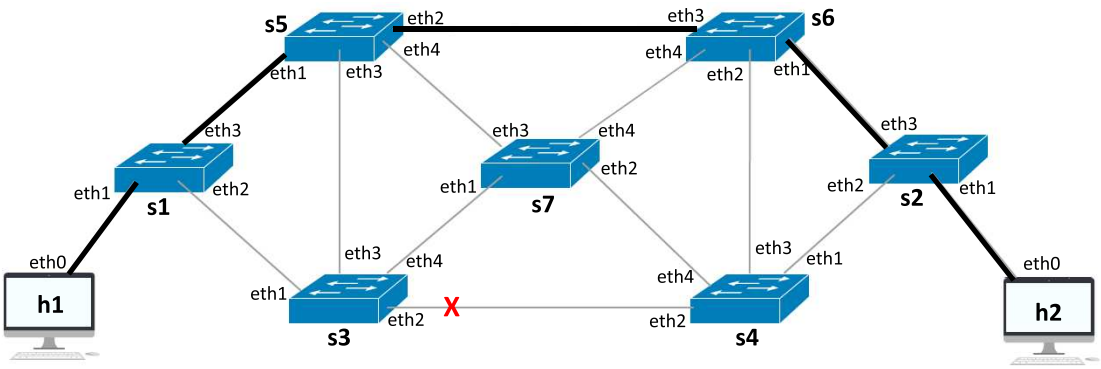

Figure 4.18: Path of the packets after rapid spanning tree protocol handled link failure

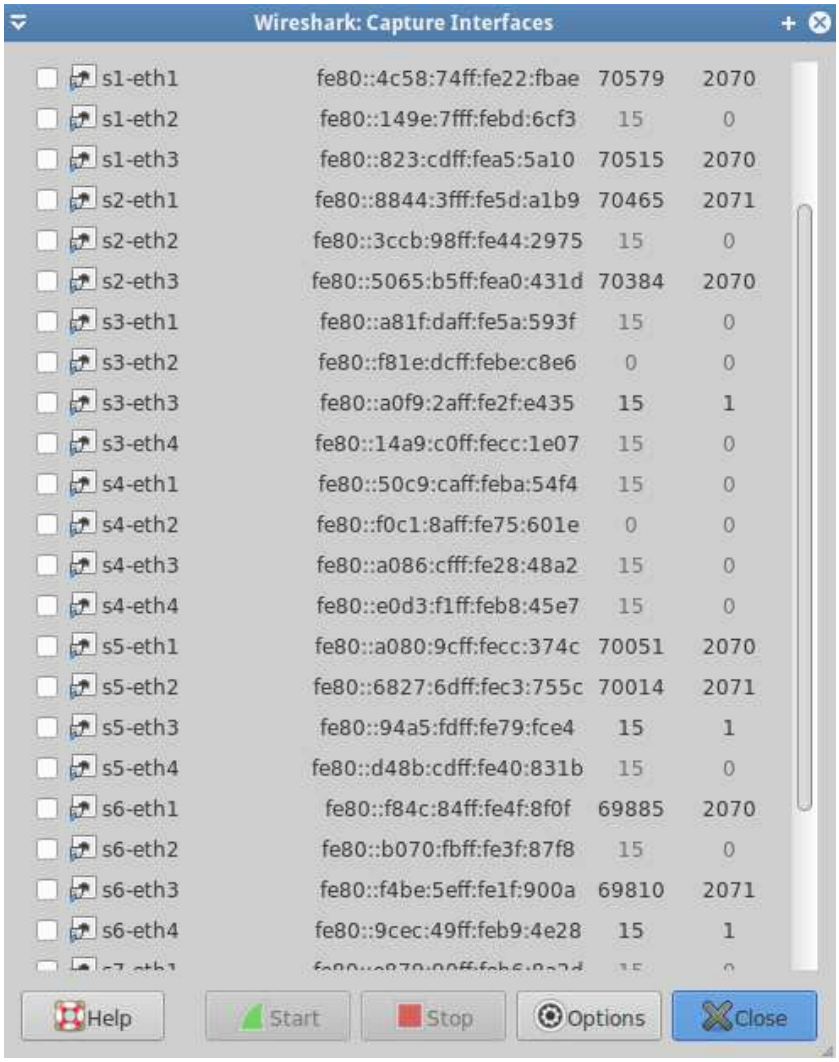

Figure 4.19: Wireshark's capture interface showing the packet counts for each port of each switch after link failure

Calculation of the time to handle link failure

We calculated the time to handle link failure in our experiment in this step. We know that the timestamp of the link failure $T_f = 1481109655.3601229$. Earlier we said that the host $h2$ keeps a log file of the received packets. We observed that log file and looked for the first packet which was received after T_f. The following subsection of the log file contains that packet.

```
Packet 22050 Received from host1 at 1481109655.3562250137! Time taken=0.0001089573 Avg=0.0007678890
Packet 22051 Received from host1 at 1481109655.3564410210! Time taken=0.0002160072 Avg=0.0004919481
Packet 22052 Received from host1 at 1481109655.3630120754! Time taken=0.0657105450 Avg=0.0035315013
Packet 22053 Received from host1 at 1481109655.3631110191! Time taken=0.0000989437 Avg=0.0018152225
Packet 22054 Received from host1 at 1481109655.3631749153! Time taken=0.0000638962 Avg=0.0009395593
```

Figure 4.20: Log file containing information about packets received by $h2$

From the above figure, we determined that the packet of index 22052 was received first after the time T_f. So, the time interval associated with that packet is the delay caused by rapid spanning tree protocol to handle the link failure, which is 0.065710545 seconds.

Summary of the experiment

The experimental result indicates that to handle link failure the rapid spanning tree protocol (RSTP) improves the performance to a great extent in comparison to spanning tree protocol (STP). From the experimental result, the delay is less than a second which is very efficient. However, for critical network infrastructure, this delay is still significant as a delay of even few milliseconds can affect the performance of the network. So, better approaches are necessary to handle link failure in critical infrastructures.

4.3 Experiment 3: Link failure delay with existing fast failover mechanism in Software Defined Networking

The objective of this experiment is to use an existing fast failover mechanism [3] for software defined networking to calculate the delay to handle link failure in the network. The existing mechanism handles link failure by installing main and backup paths using flow table entries. We showed that the fast failover mechanism can be a better approach than rapid spanning tree protocol to handle link failure in critical network infrastructures.

Description

We created a software defined network using Mininet and Ryu SDN Controller [15] and enabled the fast failover mechanism proposed in the paper [3] in the controller and then calculated the time the mechanism takes to handle link failure. The proposed mechanism calculates main and backup paths on demand when link failure happens. The data plane of the network consisted of six OpenFlow-enabled switches named from $s1$ to $s7$ and two hosts $h1$ and $h2$. The control plane of the network consists of the Ryu controller and the secured channels by which the controller is connected to the switches. On top of Ryu controller, we developed an application which implements topology detection, graph formation and the fast failover mechanism described in the paper [3]. We started the network along with the controller and then we initiated a link failure in the network by disabling a link. Finally, we have calculated the time to handle link failure in the network using the fast failover mechanism with the help of Ryu controller.

Tools and Equipment

- Mininet: A software emulator for experimenting with large networks in single machine.
- Ryu SDN Controller: An SDN controller software which can be used with Mininet.
- Open vSwitch: A software implementation of virtual switch which can act like OpenFlow enabled switches.
- Wireshark: A packet capturing and analyzing tool.

Experimental Procedure

We conducted the following steps for this experiment.

Creating the network topology

In this step, we created our experimental network topology using the API's provided by Mininet and used Ryu controller as the controller of the data plane. The data plane of the network topology consists of six Open vSwitch [20] switches

named from $s1$ to $s7$ and two hosts $h1$ and $h2$. The control plane of the topology consists of an SDN controller. The links present in the topology are bidirectional. The IP addresses of hosts $h1$ and $h2$ are 10.0.0.1 and 10.0.0.2 respectively. Using the API's provided by Mininet, we stored the ARP information of the hosts in each host. The switches are OpenFlow [1] enabled, that means the controller can control the forwarding behavior of the switches using OpenFlow protocol.

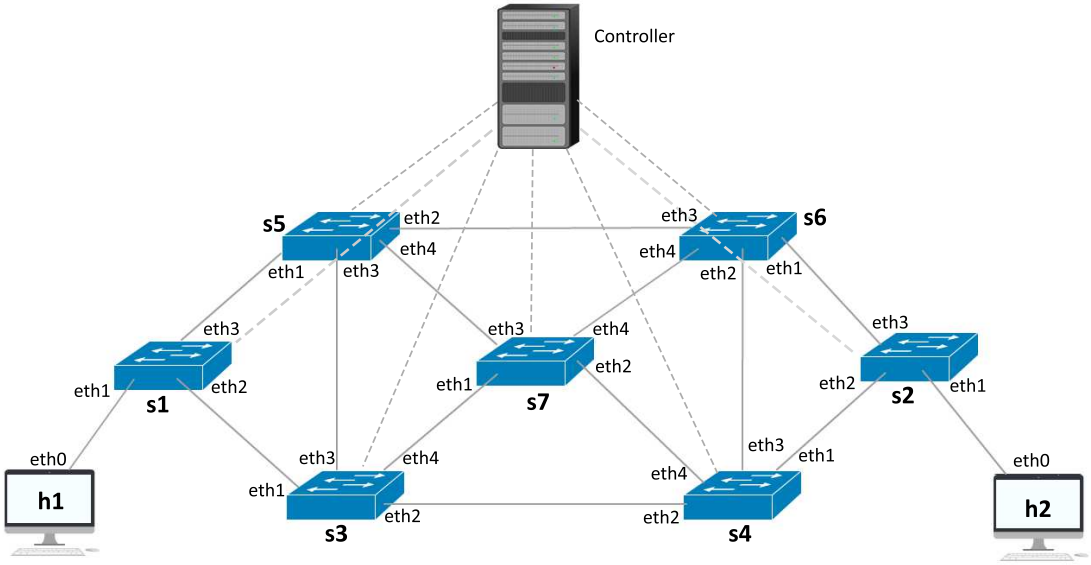

Figure 4.21: Software defined network used in the experiment

Starting the network along with the SDN controller

We ran our application on Ryu Controller from the Linux terminal. The application implemented the fast failover approach described in the paper [3] which is described in 1. After the controller initializes, the application on the controller waits for topology discovery.

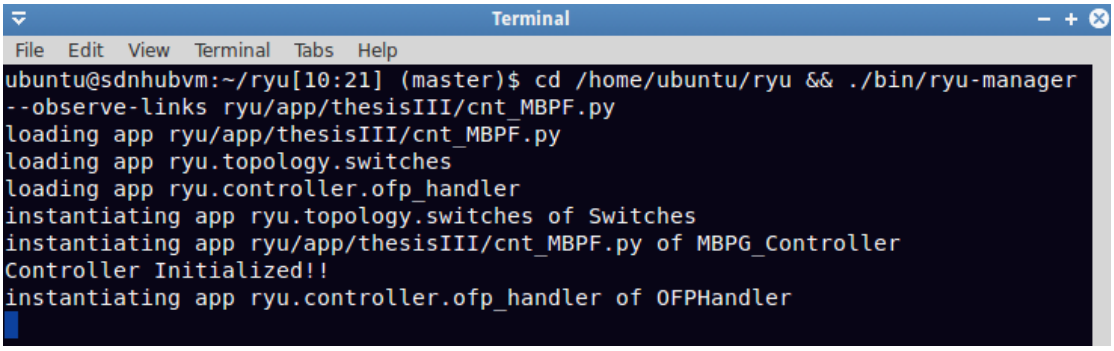

Figure 4.22: The terminal after the controller initialized.

After starting the controller, we started the data plane on Mininet terminal which is the network consisting of switches and hosts. The following is the output of the Mininet terminal.

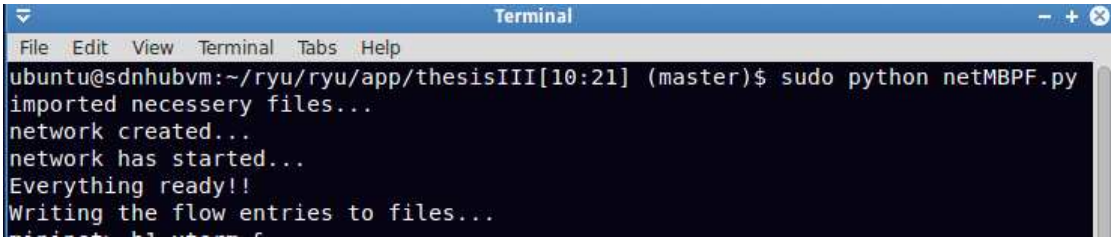

Figure 4.23: Mininet terminal after the data plane of the network was started.

After both the data plane and control plane was initialized and running, the OpenFlow enabled switches attempted to establish connections with the controller and the controller started accepting the switches after listening to "connection up" events. The application saved the datapaths of the switches and output the switches' datapath IDs. Each datapath object represents a switch and a connection of that switch with the controller. The switches continuously exchange link layer discovery protocol (LLDP) packets with each other. When a switch sends LLDP packet to a neighboring switch and then gets the reply back from that switch, it indicated the existence of a link. Then the switch sent an OpenFlow message to the controller indicating the "link up" event and thus the controller detected the link and constructed the graph.

Figure 4.24: Ryu controller detects the topology after listening to "connection up" and "link up" events.

At the end of this step, both of the data plane and control plane were initialized and ready for the experiment. The control plane learned about the topology and constructed a graph with the links in the data plane.

Running the communication between hosts in the network

In this step we made $h1$ send continuous UDP packets to $h2$ and determined the path taken by the packets by observing the flow tables of each switch. We opened two xterm windows for host $h1$ and $h2$ respectively. From xterm of $h1$ we executed script to make the host $h1$ send UDP packets to host $h2$ continuously. From xterm of $h2$ we ran another script for receiving the UDP packets from $h1$. The host $h2$ output the sequence number given to the packet, the timestamp of it's arrival and the average time on the terminal and also saved them in a log file. At first, when $h1$ tried to send the first packet to host $h2$, it forwarded the packet to the switch $s1$. However, the switch $s1$ had no idea how to forward the packet to host $h2$. So the switch $s1$ sent the packet to the controller as "packet in" message using OpenFlow through the datapath. The controller received the "packet in" messages and calculates two paths- one main path and one backup path. Then the controller installed flow table entries to the switches according to the paths calculated. The flow entries associated with the main path had priority 500 and the backup path had 300.

```
10.0.0.1 is trying to communicate with 10.0.0.2
Calculating shortest path between 10.0.0.1 and 10.0.0.2
Shortest main path [(1, 2), (3, 2), (4, 1), (2, 1)]
Installing the main path flow entries to switches...
Installing the backup path flow entries to switches...
Done!
```

Figure 4.25: Ryu controller listens to the "packet in" events and installs main and backup paths in the flow tables of the network switches

After the controller installed paths in the network, the host $h1$ could successfully send UDP packets to host $h2$. The paths were installed as flow entries in the switches. We saved all the flow table entries of each switch in text files [30]. The following is the flow table entries for the switch $s1$. It indicates that for packets with destination $h2$, the main path is the port 2 and the backup path is the port 3.

```
cookie=0x0, duration=2.014s, table=0, n_packets=2, n_bytes=196, idle_age=0,
priority=500,ip,nw_dst=10.0.0.2 actions=output:2

cookie=0x0, duration=2.009s, table=0, n_packets=0, n_bytes=0, idle_age=2,
priority=300,ip,nw_dst=10.0.0.2 actions=output:3

cookie=0x0, duration=13.068s, table=0, n_packets=16, n_bytes=1296, idle_age=1, priority=0
actions=CONTROLLER:65535
```

Figure 4.26: Flow table of the switch s1

We saved all the flow table entries of all the switches in text files. By observing the flow table entries of all the switches, we found that the packets were taking the main path $h1 \rightarrow s1 \rightarrow s3 \rightarrow s4 \rightarrow s2 \rightarrow h2$. The links $s1 \rightarrow s5$ and $s5 \rightarrow s3$ are the backup paths for the link $s1 \rightarrow s3$. The links $s3 \rightarrow s7$ and $s7 \rightarrow s4$ are the backup paths for the link $s3 \rightarrow s4$. The links $s4 \rightarrow s6$ and $s6 \rightarrow s2$ are the backup paths for the link $s4 \rightarrow s2$.

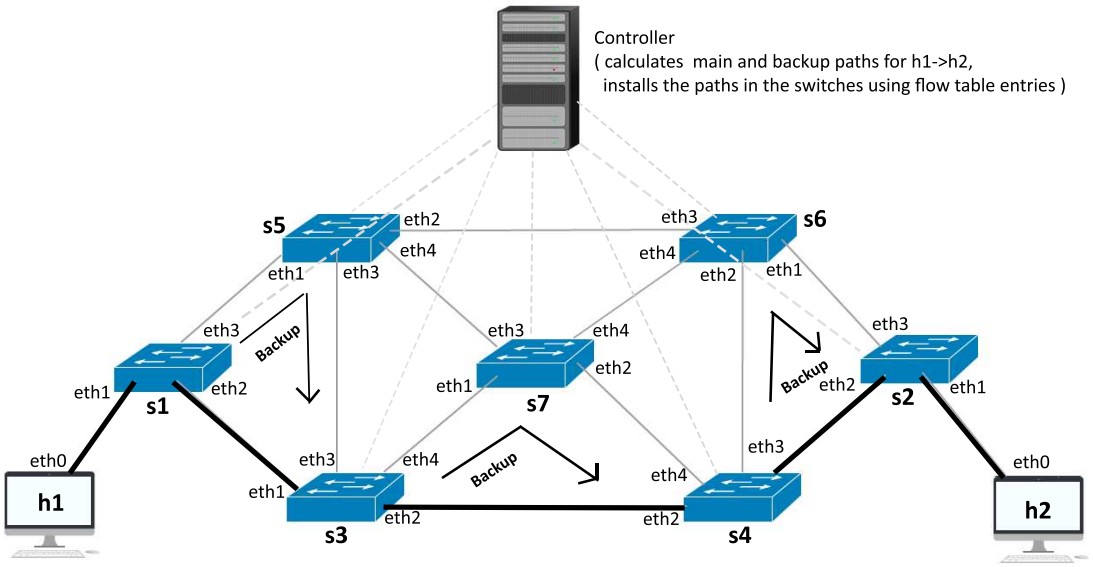

Figure 4.27: The main and backup paths for communication between $h1$ and $h2$ in the topology.

Thus, by observing the flow tables of each switch, we determined the path taken by the packets from $h1$ to $h2$ in the network at the end of this step.

Initiating link failure in the network

In this step, we initiated a link failure in the network while $h1$ was communicating with $h2$. We ran scripts to send continuous UDP packets from host $h1$ to host $h2$ in their xterm windows. For testing link failure, we ran a script which disabled the port $s3 - eth2$ in the switch $s3$. The behavior of the switches can be controlled externally in Mininet through different commands for Open vSwitches. After disabling the link, the timestamp of the link failure was shown on the terminal. From the terminal, we see that the timestamp when link failure happened is 1481110307.7036819458.

```
ubuntu@sdnhubvm:~/ryu/ryu/app/thesisIII[03:31] (master)$ sudo python linkbreaker
.py
disabling port s3-eth2
Port s3-eth2 was disabled at 1481110307.7036819458
ubuntu@sdnhubvm:~/ryu/ryu/app/thesisIII[03:31] (master)$
```

Figure 4.28: Initiation of link failure in the network

Observation

In this step, we observed the alternative path the packets from $h1$ to $h2$ were taking after the link failure happened. We determined the path by observing the modified flow tables of each switch which were saved by our script after the link failure. The flow tables can be found in [30]. In the xterm of $h2$, we observed

that the delay caused by handling the link failure was very short. No packet was lost and the path restoration occurred within a moment. The link failure was handled by the controller. The switches periodically exchanged LLDP packets to each other to detect the links. So, after the link was down, then the switch sent messages to the controller using OpenFlow which causes "link down" event in the controller. Then the controller removed all the flow entries in the switch's flow table which are associated with the failed link. So, after the main path failed and after the controller removed all the flow entries associated with it, then the flows were directed to the link associated with the backup path. In the meantime, the controller started to calculate new main and backup paths and when finished, installed them in the switches flow tables. Then the switches started to forward packets through the newly installed main path.

```
Link DOWN 3-2 -> 4-2
Handling the link failure
Removing affected flows
Calculating shortest path between 10.0.0.1 and 10.0.0.2
Shortest main path [(1, 3), (5, 2), (6, 1), (2, 1)]
Installing the main path flow entries to switches...
Installing the backup path flow entries to switches...
Done!
```

Figure 4.29: Mininet terminal showing that controller detected link failure and took necessary actions

In above figure, the output on the terminal describes how the controller reacted after detecting the link failure according to the 1. After detecting link failure, controller removed the flow entries associated with port $s3 - eth2$ of switch $s3$.

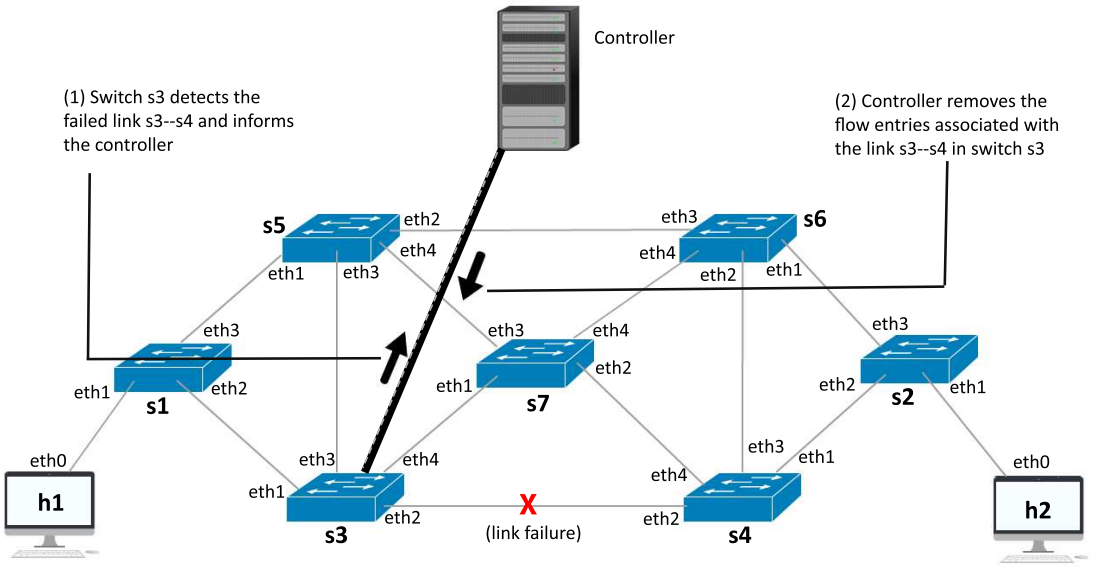

Figure 4.30: Controller detecting link failure and removing the affected flow entries

After the controller removed the affected flow entries related to the failed link, the flow of packets took the backup path stored in the network. The packets were

forwarded through the path $h1 \rightarrow s1 \rightarrow s3 \rightarrow s7 \rightarrow s4 \rightarrow s2 \rightarrow h2$. In the meantime, the controller started calculating new main and backup paths.

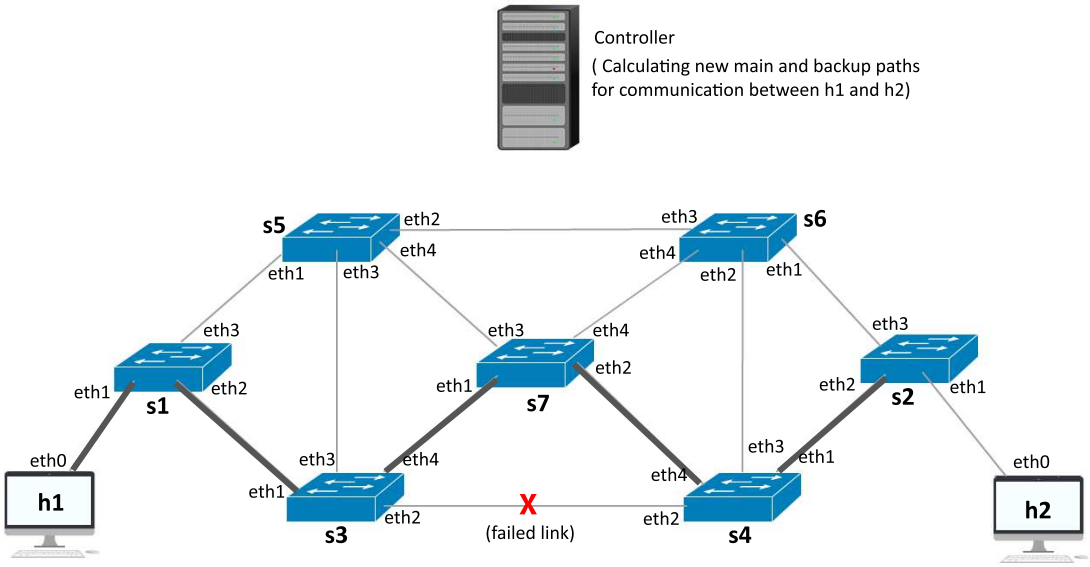

Figure 4.31: Controller starts calculating new main and backup paths, in the meantime, all the packets are forwarded through the backup path

After finishing calculation of new main and backup paths, the controller installed them in the network. We observed the modified flow tables and found that the packets were forwarded throw the new main path $h1 \rightarrow s1 \rightarrow s5 \rightarrow s6 \rightarrow s2 \rightarrow h2$.

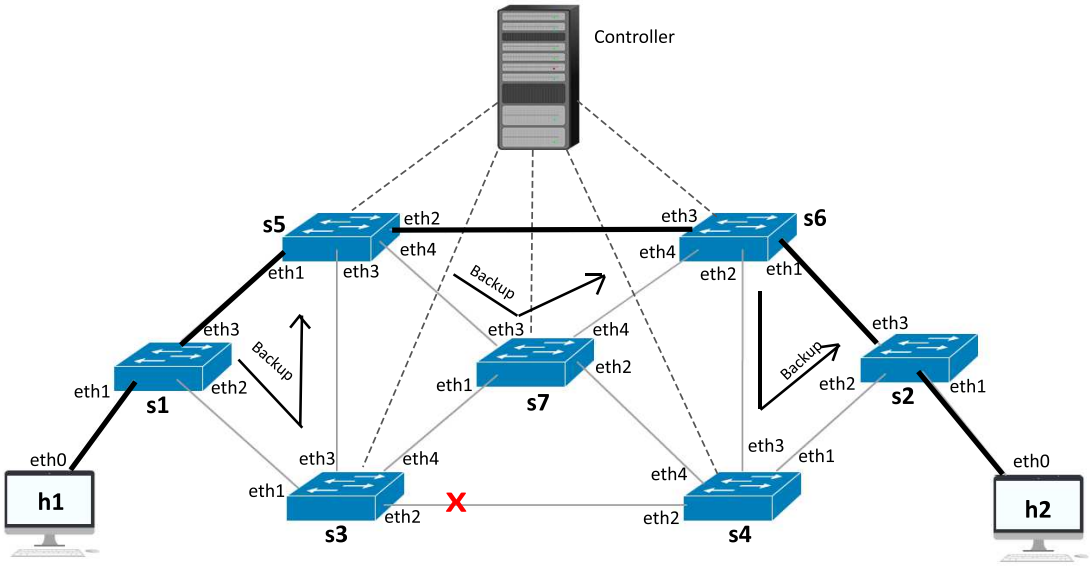

Figure 4.32: Controller installs new main and backup paths and all the packets are forwarded through the new main path

After the controller installed new main and backup path in the network, the

flow tables of the switches are changed. The change in the flow table of switch $s1$ is as follows. Here for packets with destination $h2$, the main path is port 3 and backup path is port 2.

```
cookie=0x0, duration=59.904s, table=0, n_packets=2, n_bytes=196, idle_age=79,
priority=500,ip,nw_dst=10.0.0.2 actions=output:3

cookie=0x0, duration=59.901s, table=0, n_packets=0, n_bytes=0, idle_age=81,
priority=300,ip,nw_dst=10.0.0.2 actions=output:2

cookie=0x0, duration=93.054s, table=0, n_packets=16, n_bytes=1296, idle_age=81, priority=0
actions=CONTROLLER:65535
```

Figure 4.33: Flow table of $s1$ after the controller handled link failure

Thus we have determined the change in the path of the packets by observing the flow tables of the switches after the link failure in this step.

Calculation of the time to handle link failure

From above steps, we know that the timestamp of the link failure $T_f = 1481110307.7036819458$. Earlier we said that the host $h2$ keeps a log file of the received packets. We observed that log file and looked for the first packet which was received after T_f. The following subsection of the log file contains that packet.

```
Packet 22126 Received from host1 at 1481110307.6787390709! Time taken=0.0004160404 Avg=0.0004000133
Packet 22127 Received from host1 at 1481110307.6809151173! Time taken=0.0021760464 Avg=0.0012880298
Packet 22133 Received from host1 at 1481110307.7279438972! Time taken=0.0470287800 Avg=0.0241584049
Packet 22134 Received from host1 at 1481110307.7300519943! Time taken=0.0021080971 Avg=0.0131332510
Packet 22135 Received from host1 at 1481110307.7348461151! Time taken=0.0047941208 Avg=0.0089636859
```

Figure 4.34: Part of the log file received by host h2

From the above figure, we determined that the packet of index 22133 was received first after the time T_f. The delay caused by the controller is 0.0470287800 seconds. This delay was the sum of the time taken by switches to detect the failed link and the time taken by controller to remove the flow entries associated with the failed link.

Summary of the experiment

From the experimental result we have found that the fast failover mechanism proposed in the paper [3] and described in 1 can be a more efficient approach then rapid spanning tree protocol of traditional network to handle link failure in a network. Because the delay caused by this approach is slightly less than the delay caused by rapid spanning tree protocol in the previous experiment. So we can say that for using in critical network infrastructure, the fast failover mechanism proposed in the paper [3] can be a better approach than RSTP for critical network infrastructures.

Chapter 5

Experiment 4: Link failure delay with the proposed failover recovery mechanism for Software Defined Networking

The objective of this experiment is to calculate the link failure delay in a software defined network using the failover recovery mechanism proposed in this project. The mechanism is outlined in algorithm 2 which is a modification of the mechanism proposed in the paper [3] using group table. We showed that our proposed mechanism can perform better than the existing link failure handling mechanism, hence it can be useful in critical network infrastructures.

Description

In this experiment, we used our proposed failover recovery mechanism to calculate the delay to handle link failure in software defined network. The mechanism uses fast failover type group table entries of OpenFlow [1] enabled switches and installs main and backup paths when link failure happens. We calculated the time to handle link failure using the proposed failover recovery mechanism described in algorithm 2. We created a software defined network using Mininet [13] and Ryu SDN Controller[15]. The data plane of the network consists of six switches named from $s1$ to $s7$ and two hosts $h1$ and $h2$. The control plane of the network consists of the Ryu controller and the secured channels by which the controller is connected to the switches. The communications between the controller and the data plane of the network were done using OpenFlow protocol version 1.3 which allows group table in SDN switches. We developed an application on Ryu controller which implements topology detection, graph formation and the proposed failover mechanism described in the algorithm 2. We started the network along with the controller and initiated a link failure in the network. Finally, we have calculated the time to handle link failure in the network using the proposed failover mechanism. The scripts which were used to run this experiment can be found in [31].

Tools and Equipment

- Mininet: A software emulator for experimenting with large networks in single machine.
- Ryu Controller: An SDN controller software which can be used with Mininet
- Open vSwitch: A software implementation of virtual switch which can act like OpenFlow enabled switches.
- Wireshark: A packet capturing and analyzing tool.

Experimental Procedure

The experiment includes the following steps.

Creating the network topology

In this step, we created the software defined network by our script written with the API's of Mininet and Ryu to create the data plane and the controller. We developed the experimental network topology using Mininet and Ryu controller. The data plane of the network topology consists of six switches named from $s1$ to $s7$ and two hosts $h1$ and $h2$. The control plane of the topology consists of an SDN controller. The IP addresses of hosts h1 and h2 are 10.0.0.1 and 10.0.0.2 respectively. Using the API's provided by Mininet, we stored the ARP information of the hosts in each host. The switches are enabled with OpenFlow version 1.3 because the experiment involved using group table which are the features of switches having OpenFlow 1.1 or more.

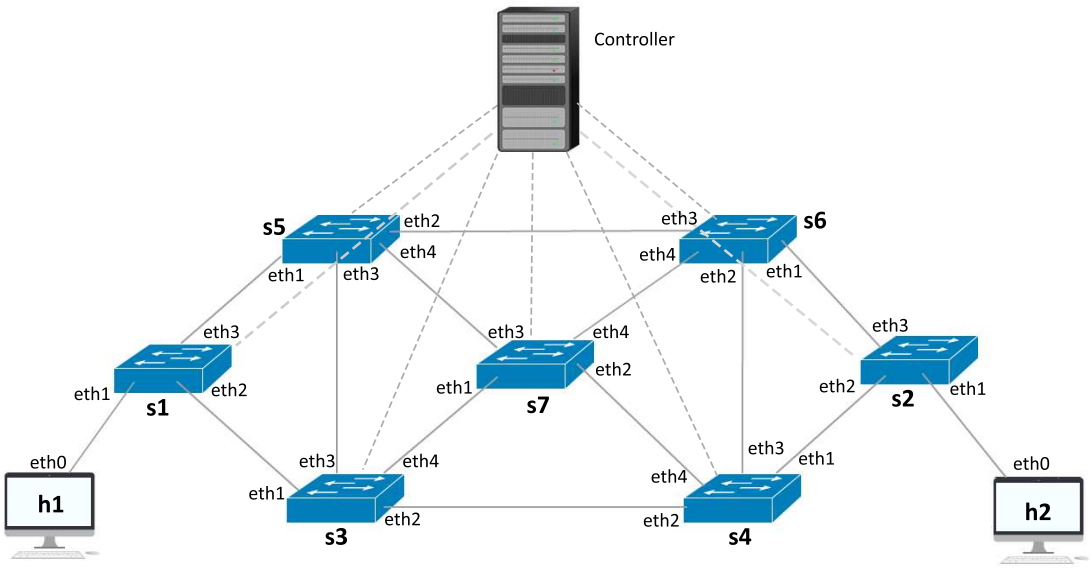

Figure 5.1: Software defined network used in the experiment

Starting the network along with the SDN controller

We ran our application on Ryu Controller from the terminal. The application implemented the failover mechanism proposed in this project which is described in the algorithm 2. At first, the controller was started. Our application which was running on the controller implemented the proposed failover mechanism. After the controller initializes, the application on the controller listens for events to detect the topology.

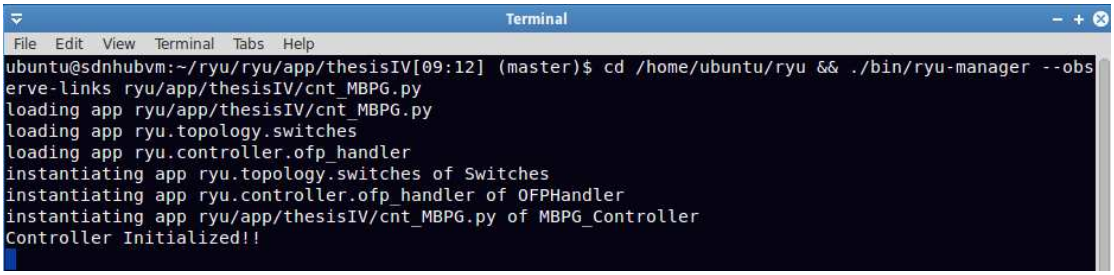

Figure 5.2: The terminal after the controller was initialized and listening for events

After the controller was started, we started the data plane on Mininet terminal. The following is the output of the Mininet terminal.

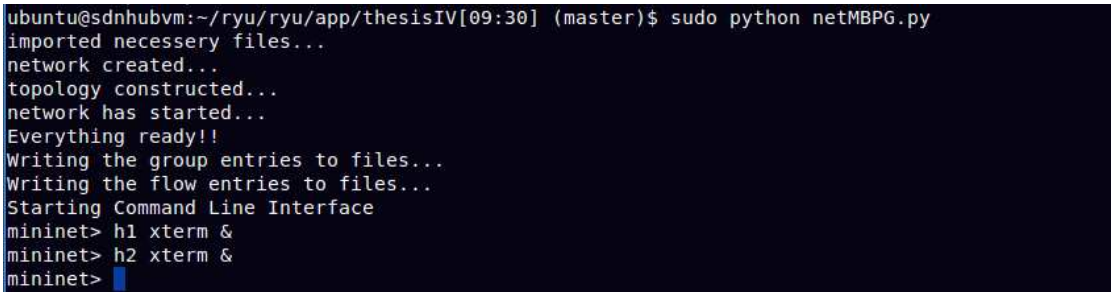

Figure 5.3: Mininet terminal after the network started running

After starting the network, the OpenFlow enabled switches attempted to establish connections with the controller and the controller started accepting the switches after listening to "connection up" events. The application saved the datapaths of the switches and output the switches' datapath IDs. The switch started detecting the links using LLDP packets and sent the link information to the controller causing the "link up" event. Controller listened to the "link up" events and detected the topology.

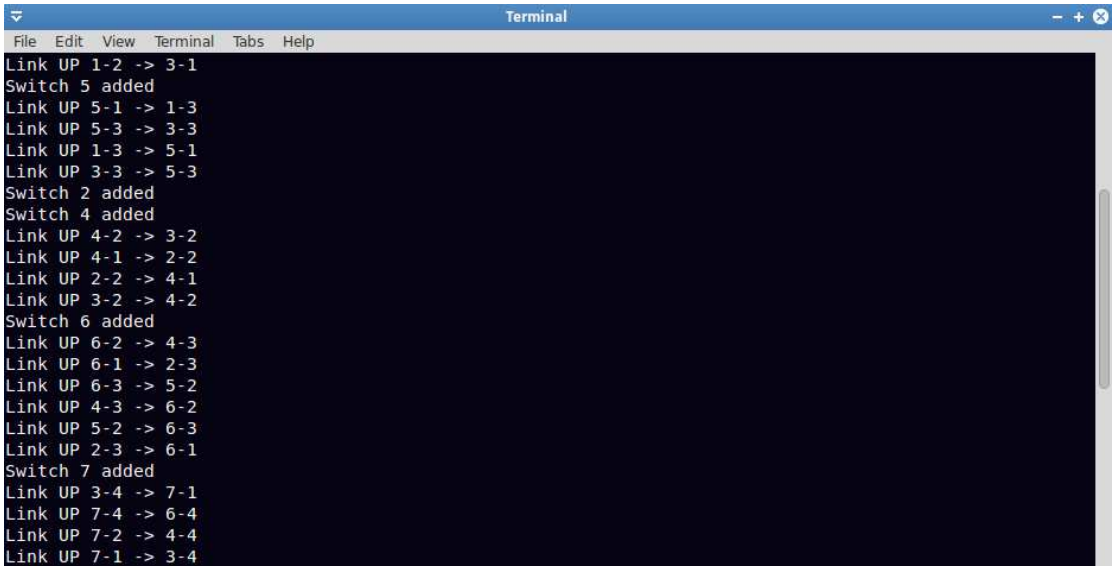

Figure 5.4: Ryu controller detecting the topology

At the end of this step, both of the data plane and control plane were initialized and ready for the experiment. The control plane learned about the topology and constructed a graph with the links in the data plane.

Running the communication between hosts in the network

In this step we made $h1$ send continuous UDP packets to $h2$ in the network and determined the path taken by the packets by observing the flow tables and group tables of each switch. We opened two xterm windows for host $h1$ and $h2$ respectively. After running our scripts the host $h1$ started to send UDP packets to $h2$ continuously and $h2$ was receiving the packets. Host $h2$ showed output on the terminal after receiving each packet and saved the outputs in text files. When $h1$ tried to send the first packet to host $h2$, it forwarded the packet to the switch $s1$. But, the switch $s1$ could not forward the packet because there were no instructions present in the flow table of $s1$ for the packet. So the switch $s1$ forwarded the packet to the controller as "packet in" message. After receiving the "packet in" message, the controller calculated two paths- one main and one backup path. The controller constructed a group table entry of type "fast failover". There were two action buckets in the group entry, the first one indicated the main path and the second one indicated the backup path. The controller then installed the group entry in the group table of the switch $s1$. The controller also installed a flow table entry in the flow table of switch $s1$, which applies actions on the packets with destination $h2$ according to the installed group entry.

```
10.0.0.1 is trying to communicate with 10.0.0.2
Calculating shortest path between 10.0.0.1 and 10.0.0.2
Shortest main path [(1, 2), (3, 2), (4, 1), (2, 1)]
Installing the flow and group entries to switches...
Adding Group entry 1 to switch1
Adding Group entry 1 to switch3
Adding Group entry 1 to switch4
Done!
```

Figure 5.5: Ryu controller listens to the "packet in" events and installs main and backup paths as group table entries

After the controller installed main and backup paths, the host $h1$ could successfully send packets to host $h2$. The following figures describe the group table and flow table of the switch $s1$. The group table contains a fast failover group entry which indicates that the packets with destination $h2$ will be forwarded through the port $s3 - eth2$ if the port $eth - 2$ is alive. If the port $eth - 2$ is not alive, the packets will be forwarded through the port $eth - 3$ if the port $eth - 3$ is alive. The flow table contains a flow entry which applies actions on the packet with destination $h2$ according to the group table entry described before. The flow and group table entries of each switch are saved in text files [32]. The following figure shows the group table of switch $s1$.

```
group_id=1,type=ff,bucket=weight:0,watch_port:2,actions=output:2,bucket=weight:0,watch_port:3,acti
ons=output:3
```

Figure 5.6: Group table of the switch s1

And the following figure shows the flow table of the switch $s1$.

```
cookie=0x0, duration=2.094s, table=0, n_packets=2, n_bytes=196, idle_age=0,
priority=500,ip,nw_dst=10.0.0.2 actions=group:1

cookie=0x0, duration=13.246s, table=0, n_packets=17, n_bytes=1366, idle_age=1, priority=0
actions=CONTROLLER:65535
```

Figure 5.7: Flow table of the switch s1

Earlier we described that we saved all the flow table and group entries of all the switches in text files. By observing the flow and group tables of all the switches, we found that the packets are taking the main path $h1 \rightarrow s1 \rightarrow s3 \rightarrow s4 \rightarrow s2 \rightarrow h2$. The links $s1 \rightarrow s5$ and $s5 \rightarrow s3$ are the backup paths for the link $s1 \rightarrow s3$. The links $s3 \rightarrow s7$ and $s7 \rightarrow s4$ are the backup paths for the link $s3 \rightarrow s4$. The links $s4 \rightarrow s6$ and $s6 \rightarrow s2$ are the backup paths for the link $s4 \rightarrow s2$.

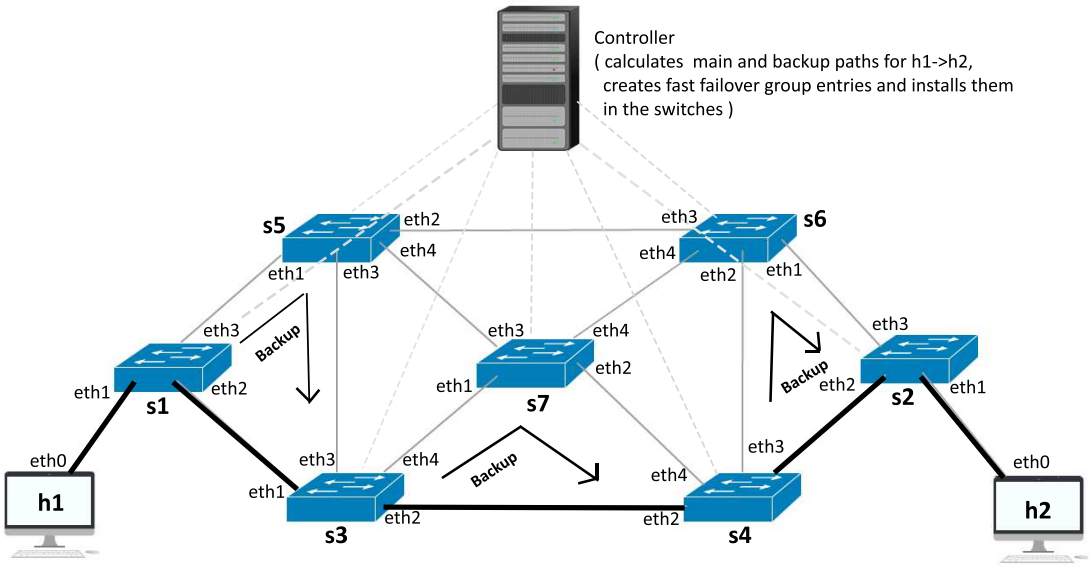

Figure 5.8: The main and backup paths for communication between $h1$ and $h2$ in the topology

Thus, by observing the flow tables and group tables of each switch, we determined the path taken by the packets from $h1$ to $h2$ in the network at the end of this step.

Initiating link failure in the network

In this step, we initiated a link failure in the network while $h1$ was communicating with $h2$. We ran scripts to send continuous UDP packets from host $h1$ to host $h2$ in their xterm windows. For testing link failure, we ran another script which disabled the port $s3 - eth2$ in the switch $s3$. After disabling the link, the timestamp of the link failure was shown on the terminal. From the terminal, we see that the timestamp when link failure happened is 1481110860.26883792.

```
ubuntu@sdnhubvm:~/ryu/ryu/app/thesisIV[03:38] (master)$ sudo python linkbreaker.
py
disabling port s3-eth2
Port s3-eth2 was disabled at 1481110860.2688379288
ubuntu@sdnhubvm:~/ryu/ryu/app/thesisIV[03:41] (master)$
```

Figure 5.9: Initiation of link failure in the network

Observation

In this step, we observed the alternative path the packets from $h1$ to $h2$ were taking after the link failure happened. We determined the path by observing the modified flow tables and group tables of each switch which were saved at runtime after the link failure. The flow tables and group tables can be found in [32]. In

the xterm of $h2$, we observed that the delay caused by handling the link failure was almost negligible. No packet was lost and the path restoration occurred within a moment. The link failure was handled by the controller. As switches exchange LLDP packets periodically, they detected the failed link and informed the controller. At the same time, the flow of the packets switched locally to the backup path. The local switching occurred because the fast failover feature of the group table entry allowed it to do so. So the flow was not interrupted at all. The controller calculated new main and backup paths and replaced them later in the group table.

```
Link DOWN 3-2 -> 4-2
Handling the link failure
Calculating shortest path between 10.0.0.1 and 10.0.0.2
Shortest main path [(1, 3), (5, 2), (6, 1), (2, 1)]
Installing the flow and group entries to switches...
Adding Group entry 2 to switch1
Adding Group entry 2 to switch5
Adding Group entry 2 to switch6
Done!
```

Figure 5.10: Mininet terminal showing that controller detected link failure and took necessary actions

The above figure shows a terminal where the controller reacted after detecting the link failure. As soon as the link associated with main path failed, the flow of packets were switched to the backup path $h1 \rightarrow s1 \rightarrow s3 \rightarrow s7 \rightarrow s4 \rightarrow s2 \rightarrow h2$. In the meantime, the controller started calculating new main and backup paths.

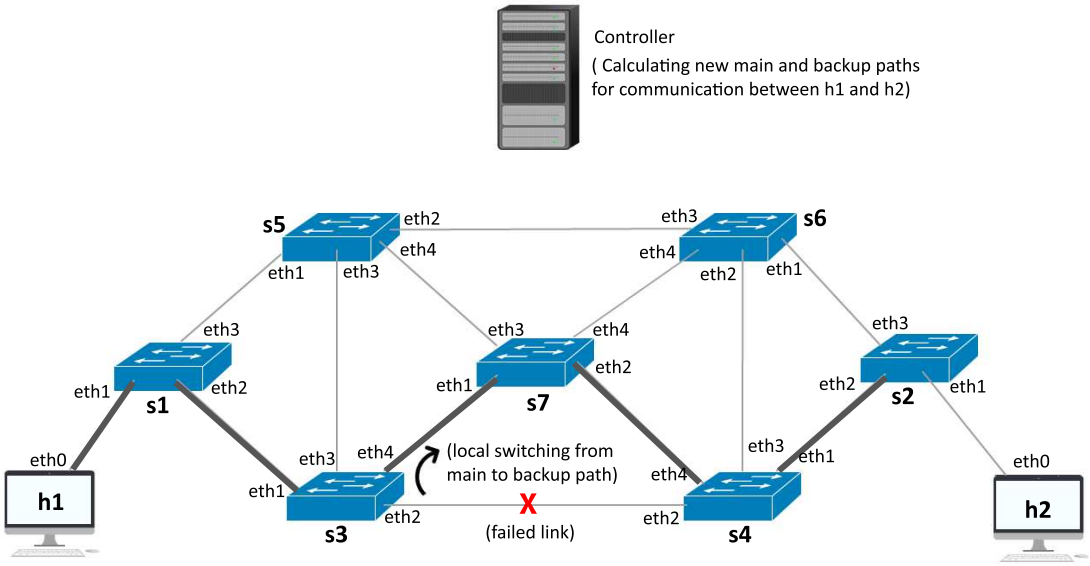

Figure 5.11: Flow of packet switched locally after the link failure and forwarded through the backup path

After finishing calculation of new main and backup paths, the controller installed them in the network switches as fast failover type group table entries where

the main path is associated with the first bucket and the backup path is with the second bucket. We observed the flow tables and group tables and found that the packets were forwarded throw the new main path $h1 \rightarrow s1 \rightarrow s5 \rightarrow s6 \rightarrow s2 \rightarrow h2$.

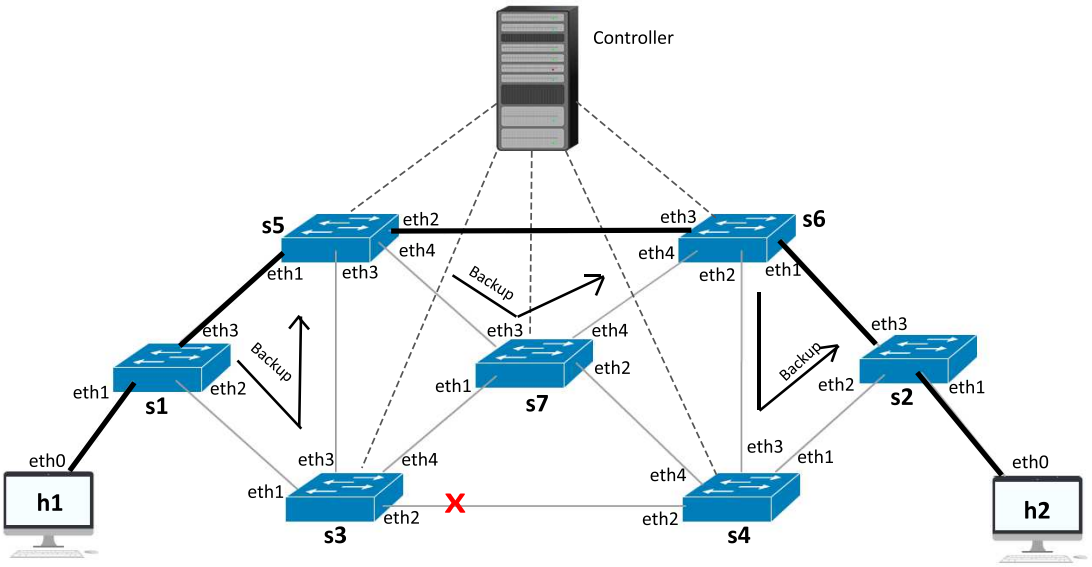

Figure 5.12: Controller installs new main and backup paths and all the packets are forwarded through the new main path

After installing new main and backup path in the network, our script saved the modified flow and group tables of all the switches in text files. From the saved flow tables and group tables, the modified group table of switch $s1$ is bellow.

```
group_id=2,type=ff,bucket=weight:0,watch_port:3,actions=output:3,bucket=weight:0,watch_port:2,acti
ons=output:2
```

Figure 5.13: Modified group table of the switch $s1$

And the modified flow table of switch $s1$ is bellow:

```
cookie=0x0, duration=3.424s, table=0, n_packets=2, n_bytes=196, idle_age=10,
priority=500,ip,nw_dst=10.0.0.2 actions=group:2

cookie=0x0, duration=23.870s, table=0, n_packets=17, n_bytes=1366, idle_age=12, priority=0
actions=CONTROLLER:65535
```

Figure 5.14: Modified flow table of the switch $s1$

Thus we have determined the change in the path of the packets by observing the flow and group tables of the switches after the link failure in this step.

Calculation of the time to handle link failure

From above steps, we know that the timestamp of the link failure $T_f = 1481110860.26883792$. Earlier we said that the host $h2$ keeps a log file of the received packets. We observed that log file and looked for the first packet which was received after T_f. The following subsection of the log file contains that packet.

```
Packet 28325 Received from host1 at 1481110860.2684609890! Time taken=0.0002920628 Avg=0.0003185464
Packet 28326 Received from host1 at 1481110860.2687859535! Time taken=0.0003249645 Avg=0.0003217555
Packet 28327 Received from host1 at 1481110860.2690749168! Time taken=0.0002889633 Avg=0.0003053594
Packet 28328 Received from host1 at 1481110860.2693619728! Time taken=0.0002870560 Avg=0.0002962077
Packet 28329 Received from host1 at 1481110860.2696690559! Time taken=0.0003070831 Avg=0.0003016454
```

Figure 5.15: Part of the log file received by host $h2$

From the above figure, we determined that the packet of index 28327 was received first after the time T_f. The delay caused by the controller is 0.0002889633 seconds. This delay was caused by the switching time from first action bucket of the main path to the second action bucket of backup path.

Summary of the experiment

From the experimental result, we have found that the link failure delay caused by our proposed failover recovery mechanism described in the algorithm2 is less than a millisecond which is almost negligible. The delay is shorter than all the experiments with existing mechanisms in previous mechanisms. So, this mechanism can be a good choice for using in critical network infrastructures.

Chapter 6

Comparative Analysis

In the experiments of previous chapters we have tested different mechanisms for handling link failure in both traditional and software defined networks. From the results obtained from the experiments, we constructed the following table.

Experiment	Mechanism	Time to handle link failure
Exp 1	Spanning Tree Protocol	49.6053369 seconds
Exp 2	Rapid Spanning Tree Protocol	0.065710545 seconds
Exp 3	Fast failover mechanism 1 from [3]	0.0470287800 seconds
Exp 4	Proposed failover recovery mechanism 2	0.0002889 seconds

Our first and second experiments were about testing the delay of link failure in the traditional network. In the first experiment, we enabled the traditional network with spanning tree protocol caused a significant delay of almost 49 seconds. The delay was caused by the spanning tree protocol to detect the failed link and re-converge with the network. This delay is very significant and it makes the spanning tree protocol unusable in the critical infrastructures. Using rapid spanning tree protocol in the second experiment, the delay was improved to a significant extent as the delay was only 65 milliseconds. This might be a great improvement, but for critical infrastructures, the delay needs to be even shorter because a few milliseconds of delay can lower the performance of critical infrastructures by a great extent.

Our third experiment was a software defined networking approach to handle link failure using the mechanism in the paper [3]. The 1 described in the paper [3] calculated on demand main and backup paths in case of link failure, which caused a delay of 40 milliseconds. The delay was the sum total of the delay to detect the failed link by switches, the delay to inform the controller and the delay for removing the affected flow entries from the switch by the controller. We can say that the delay is shorter than the first and second experiments, but it's still significant for critical network infrastructures. In the fourth experiment, the use of our proposed failover mechanism was used to calculate link failure delay in software defined networks. The proposed mechanism 2 caused a delay of less than a millisecond in handling the link failure, which is only 0.2 milliseconds. This is a huge improvement as the delay is almost negligible and much shorter than the results of the previous experiments.

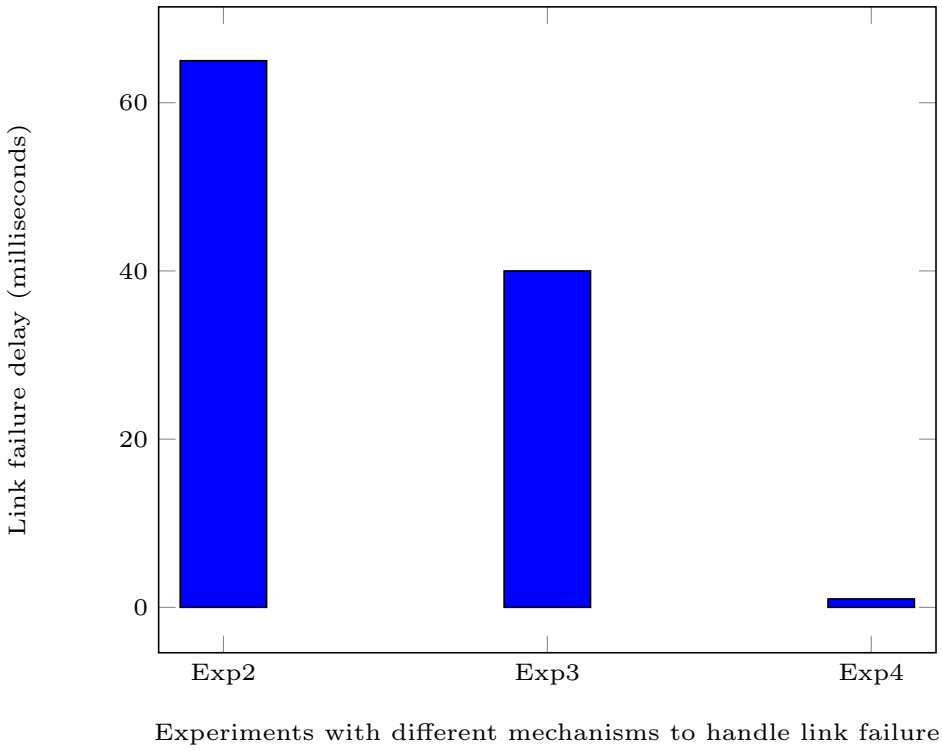

Figure 6.1: Comparison of link failure delay from experiments.

In above chart, the X-axis contains the indicators of experiment 2,3 and 4 and the Y-axis contains their corresponding results. In X-axis, $Exp2$ indicates the result of experimenting with RSTP, $Exp3$ is the result of experimenting with the mechanism of paper [3] and $Exp4$ is the result of our proposed failover recovery mechanism. The experiment with STP is not added to the graph as it's result is almost 49000 milliseconds which is too large for the graph. From the above chart, we observe that experiment 2 has the higher delay than experiment 3 and 4. However $Exp4$ which represents the result of our proposed failover recovery mechanism has minimized the delay to handle link failure to a great extent and the delay is almost negligible comparing with the results of $Exp2$ and $Exp3$.

We conducted these four experiments multiple times and found almost same results. Since the delay is measured in milliseconds, the experiments might give a bit different results at different times, but they always maintained the same decreasing order as shown in the chart above. Therefore we can conclude that our proposed mechanism can perform better than the existing failover mechanisms.

Chapter 7

Conclusion

7.1 Project Summary

Ensuring an uninterrupted network communication during link failure is always a challenging issue in networking where link failure is a common and unpredictable event. This becomes more challenging in case of critical networks where an interruption of few milliseconds can degrade the performance severely. To keep the services uninterrupted and constant, in our project we have introduced a software defined networking approach for handling link failure in critical network infrastructures. Our works include experimenting with different mechanisms which handle link failure and performing a comparative analysis of the experimental results. We calculated link failure delays of different spanning tree protocols of traditional network and an existing fast failover mechanism for software defined networks and thus we showed that the software defined networking can provide a better solution than traditional networks in handling link failures. Moreover, we improved the existing fast failover mechanism by modifying the mechanism using group table feature of software defined networking. After experimenting with our proposed mechanism, we observed that it performs better than the existing mechanism and minimizes the link failure delays significantly. From the experimental results, we concluded that our proposed failover recovery mechanism can be more efficient for critical network infrastructures as well as for any network systems intolerant of delays.

7.2 Limitations and Future Works

From the experimental results, we found that our proposed failover recovery mechanism performs more efficiently than the existing fast failover mechanisms of software defined networks and spanning tree protocols of traditional networks. However, there exist certain limitations in our proposed mechanism. The situations where the performance of our proposed mechanism will be hampered are described bellow.

- **Multiple link failure:** In the proposed mechanism, the controller installs one main and one backup path in the data plane. When the main path fails the packets can switch to the backup path. But when both main and backup paths fail simultaneously, there will not be any paths to forward the

packets. The flow of packets will stop until the SDN controller calculates and installs new paths in the switch. Thus the performance of the network will be lowered.

- **Network congestion:** According to our proposed mechanism, the data plane will continue to forward the traffic through the main path as long as the ports related to the main path is alive. When congestion occurs in any link existing in the main path, the data plane will still use the main path to forward the network packets, hence the performance of the network will be hindered.

In future works, we will try to make our proposed failover recovery mechanism more efficient and overcome the limitations described above. The first limitation, which is the case of multiple link failure, can be solved using multiple backup paths. The SDN controller can install N-backup paths using action buckets of fast failover group table entries to the data plane so that in case of multiple link failure, the flow can be uninterrupted. To implement this N-backup paths solution, we will try to improve our proposed algorithm so that it can calculate one main path and N-backup paths. Moreover, we will enrich our proposed mechanism by adding congestion aware mechanism so that in case of congestion the network traffic can be forwarded using alternative paths. In this way, we may improve the proposed failover recovery mechanism of this project in our future works which will be useful for handling link failure efficiently in critical network infrastructures.

References

[1] *OpenFlow Switch Specification 1.3.1.* Open Networking Foundation, 2012.

[2] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in openflow networks," in *Design of Reliable Communication Networks (DRCN), 2011 8th International Workshop on the*, pp. 164–171, IEEE, 2011.

[3] N. Sahri and K. Okamura, "Fast failover mechanism for software defined networking: Openflow based," in *Proceedings of The Ninth International Conference on Future Internet Technologies*, p. 16, ACM, 2014.

[4] "Software defined networking architecture." `http://wikibon.org/wiki/v/Networking_Revolution:_Software_Defined_Networking_and_Network_Virtualization`. Accessed on 08.02.2017.

[5] "Bridge protocol data unit." `https://en.wikipedia.org/wiki/Bridge_Protocol_Data_Unit`. Accessed on 08.02.2017.

[6] "Address resolution protocol (arp)." `http://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/arp.html`. Accessed on 08.02.2017.

[7] "How spanning tree protocol (stp) prevents cycles and handles link failure in traditional network." `http://www.dummies.com/programming/networking/cisco/spanning-tree-protocol-stp-introduction/`. Accessed on 06.02.2017.

[8] "Understanding rapid spanning tree protocol (802.1w)." `http://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/24062-146.html`. Accessed on 06.02.2017.

[9] "Broadcast storm (broadcast radiation)." `https://en.wikipedia.org/wiki/Broadcast_radiation`. Accessed on 08.02.2017.

[10] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[11] T. A. Dave, "Openflow: Enabling innovation in campus networks," 2014.

[12] "Openflow." `http://archive.openflow.org/wp/learnmore/`. Accessed on 08.02.2017.

[13] "Mininet, an instant virtual network on your laptop (or other pc)." http://mininet.org/. Accessed on 06.02.2017.

[14] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, "The design and implementation of open vswitch.," in *NSDI*, pp. 117–130, 2015.

[15] "Ryu component-based software defined networking framework." https://github.com/osrg/ryu. Accessed on 06.02.2017.

[16] "Introduction to wireshark." https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html. Accessed on 06.02.2017.

[17] Y.-D. Lin, H.-Y. Teng, C.-R. Hsu, C.-C. Liao, and Y.-C. Lai, "Fast failover and switchover for link failures and congestion in software defined networks," in *Communications (ICC), 2016 IEEE International Conference on*, pp. 1–6, IEEE, 2016.

[18] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 121–126, ACM, 2014.

[19] "Fast failover group table entry." http://flowgrammable.org/sdn/openflow/message-layer/groupmod/#tab_ofp_1_1. Accessed on 07.02.2017.

[20] "Open vswitch." http://openvswitch.org/. Accessed on 06.02.2017.

[21] "ovs-ofctl - administer openflow switches." http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt. Accessed on 17.02.2017.

[22] https://github.com/Raufoon/UndergradProject2016/tree/master/exp1. Accessed on 07.02.2017.

[23] https://github.com/Raufoon/UndergradProject2016/blob/master/exp1/netSTP.py. Accessed on 07.02.2017.

[24] "xterm." https://en.wikipedia.org/wiki/Xterm. Accessed on 07.02.2017.

[25] https://github.com/Raufoon/UndergradProject2016/blob/master/exp1/sender.py. Accessed on 07.02.2017.

[26] https://github.com/Raufoon/UndergradProject2016/blob/master/exp1/receiver.py. Accessed on 07.02.2017.

[27] https://github.com/Raufoon/UndergradProject2016/blob/master/exp1/log.txt. Accessed on 07.02.2017.

[28] https://github.com/Raufoon/UndergradProject2016/blob/master/exp1/linkbreaker.py. Accessed on 07.02.2017.

[29] https://github.com/Raufoon/UndergradProject2016/tree/master/exp2. Accessed on 07.02.2017.

[30] https://github.com/Raufoon/UndergradProject2016/tree/master/exp3/Observation/tables. Accessed on 07.02.2017.

[31] https://github.com/Raufoon/UndergradProject2016/tree/master/exp4. Accessed on 07.02.2017.

[32] https://github.com/Raufoon/UndergradProject2016/tree/master/exp4/Observation/Tables. Accessed on 07.02.2017.